C From Scratch

Dan Rohr

December 26, 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 having enough experience and kowledge 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: a curious student looking to get started with a new kind of programming.

To that end, this is not a beginning programming guide. I expect anyone reading this to at least have some coding experience, ideally in another statically-typed language (like, say, Java), though experience in dynamically-typed languages like Python and MATLAB is also ok (static typing will 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.

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. Stack Overflow is out there, `cplusplus.com` is out there. Use them. Some of it might be nontrivial to figure out, depending on your level of experience, but the experience of figuring it out will ultimately be worth it.

What I ask 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 (if I did my job) know what you need to know to move forward, and eventually get a formidable knowledge of this language.

I should mention that this is not the way I, or anyone I know, learned C. I just kind of got thrown into the deep end and had to kind of figure things out as I went, so things like preprocessor directives, the linker, even a lot of language features were black magic until too long ago. This, however, is going to be a bottom-up explanation: we're going to start with primitve data types and functions and build the language up from there, with as little "ignore this syntax for now" as possible. So if at any point this seems hard, it's because you're very rapidly acquiring a knowledge base that took me a long time to put together.

By the way, in the way of code style in the examples, I did my best to follow the Linux Kernel style guidelines, cited at the end of the book. A lot of the style choices might seem a little strange to someone who hasn't seen them before, but I default to Linus to justify them.

Exercises (sort of)

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**. Also, choose at least 512MB RAM. The more memory you pick, the better your performance will be, but picking more than half your machine's physical RAM is not a good idea. 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. If you've never used a Unix-like interface before, go to `http://www.ee.surrey.ac.uk/Teaching/Unix/` and read at least the first two tutorials. Now that you have your install, you'll be able to play around and get a feel for things as you read.

   From here on, I'm going to be assuming you're on a Unix-like system, and have some very basic knowledge about how to get around. Make sure you know how to **C**hange **D**irectories and **LiS**t files.

3. 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
   ```

   Git is a kind of program called a version control system. Software engineers use it to keep track of the history of a project and collaborate with colleagues. Git is used pretty much accross the board in software engineering these days, and learning it is definitely a good idea for any programmer, but I don't require it for this text. If I want you to do any git commands, I'll tell you exactly what to type in.

If you'd like to learn more, check out `try.github.io` for a 15-minute interactive intro. It's actually quite nice.

4. 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.

5. 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.

6. 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.

By the way, you're obviously free to skip any of the exercises in this book. It's not like anybody's going to know, first of all. Also, there's no shame in taking a break from wrangling a segfault to learn some new stuff. But I really tried to make the exercises relevant, and I think they're all worth doing. But at the end of the day, it's up to you.

Chapter 1

The Basics

1.1 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 the first two. These concern you, the programmer. You’ll find that 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, efficient programs. It also means you have the freedom to write garbage, nigh-undebuggable code.

1.2 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. The former generally refers to behavior that the C standard simply does not specify. For example, the order in which function arguments are evaluated is unspecified. [3]

Undefined behavior, however, generally refers to an error. For example, we'll talk about arrays later on, but one common source of bugs in C (and every other language) is accessing 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.[1]

Why is this so? Why wouldn't we want that exception? 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 performance hit. Instead, the onus is on **you**, the programmer, to ensure that overflows do not happen.

Going back to the ANSI rationale, "A standard is a treaty between implementor and programmer." If both parties hold up their end of the treaty, everything works great. If not, you've got issues. You might want to go and read the introduction to the ANSI rationale. It's not long and pretty informative.

[1] In fact, the worst thing that could happen when you have undefined behavior is your program running perfectly because of some coincidence. Then, somebody innocuously changes some compiler options, the coincidence goes away, everything breaks, and nobody can figure out why.

1.3 Data types

Enough philosophical bullshit, it's time to talk about code. C is a statically-typed language, and many of its primitive data types should be familiar from other languages. You have stuff like `int`, `long`, `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 technically be wrong. However, the way you usually think of a char in a low-level language is more along the lines of "one byte of data." 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.

Having said that, I should say at this point that the C standard, to my knowledge, does not specify the **exact** size of any datatype. What the spec demands is the **minimum** size of each datatype. So an implmentation could make `char` 64 bits and still conform to the spec. This is why the `sizeof` operator is so important. For the purposes of the beginning of this text, you use it on a type name and it returns the size in bytes of that datatype. That said, implementations will usually keep certain sizes, like `char`, to the minimum, so I apologize in advance for getting sloppy and referring to the sizes of data as if they're set in stone.

While we're talking about sizes, a notable difference from Java is that the various kinds of integers are different sizes. Plain old `int` is at least 16 bits, `long` is at least 32 bits, and `long long` is at least 64 bits. This is different from Java, where `int` is 32 bits and `long` is 64 bits. This kind of thing is how you know you're coding in a language designed in 1972, not 1995. A 64 bit integer was a ton of data back then, so the obtuse syntax of `long long` was not so bad. Now, 64 bits is a common processor word size. That said, `sizeof(int)` is 4, and `sizeof(long)` is 8 on my 64-bit Linux OS, compiling with `gcc`. 16-bit integers really don't make sense on modern machines, and language implementers understand that. Part of the beauty of the spec is that it gives freedom for the language to work both in 1972 and 2014.

Another thing to keep in mind is that there is no explicit boolean type. Instead, you use `int` to represent a boolean value, with 0 being `false` and anything else being `true`, for example:

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

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

The usual comparison operators (==, <, <=, etc), as well as the boolean logic operators, return ints for this reason.

1.4 Functions

You should be familiar with the concept of a function both from mathematics, and from other programming languages. It should be pretty easy to figure out C's function syntax. Bear in mind that all C code must be part of a function. Though you can have global variables, you can't have code (function calls, control structures, etc) sitting at top level in a C file.

A major 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

```c
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.

1.5 Declaration, Definition & Scope

Like in other statically-typed languages, data must be declared in C. This is essentially letting the compiler know that you intend to have data of a certain type. For example, if you do

```c
int i;
```

You are telling the compiler “I intend to use a value of type int, which I will identify with the keyword i.” At this point, you have declared the variable, but you have not yet defined its value. To do that, you could do[2]

```c
int i;
// other stuff, perhaps
i=0xdeadbeef;
```

You need to define a variable before you can read its value, so

```c
int i;
printf("i=%d\n", i); // ?
```

Will print an undefined number. Note that “undefined number” does not necessarily mean 0. It could be anything, C makes no promises. Finally, if you want to declare and define a variable at the same time, you can do

```c
int i=0;
printf("i=%d\n", i);// prints 0
```

What might be a little less familliar is declaring and defining functions. I mentioned earlier that order of functions in a file matters in C. The way you get around that is by declaring your function, just like how you declare a variable. Say you wanted to do the following.

```c
int fourth(int x)
{
  return square(square(x));
}

int square(int x)
{
  return x*x;
}
```

You'd get a compiler error, since when the compiler is reading `fourth`, it doesn't know that `square` exists. You could get around this by either switching the order of the functions (duh) or by declaring `square` as follows.

[2] If you're not familliar with the hexadecimal representation of numbers, go read up on that now. It's not as complicated as it looks, and it's too important not to learn about.

```
int square(int x);

int fourth(int x)
{
  return square(square(x));
}

int square(int x)
{
  return x*x;
}
```

We'll go into much more detail on how this works in 4.1.

Scope

As you might be familiar with from other languages, variables have **scope** in C. The main distinction to be made is between local scope and global scope. If you declare a variable at the "top level" of a file, its scope is called global. It can be referenced anywhere in your program, for the entire lifetime of that program. The C standard refers to this as "static storage duration."

On the other hand, if you declare a variable within a function or some other block of code, that variable's scope is local to that block. It can only be referenced within that block, and it can only be referenced while that block is still relevant. So for example, once a function call exits, that function's local variables are out of scope, and attempts to reference them will result in undefined behavior. This also applies to blocks for conditionals, loops, etc. So if you declare a variable in the `if` block of a conditional, its scope is that block. This kind of storage is called "automatic storage duration."

There is an exception to these rules, however. If you declare a local variable with the `static` keyword, you essentially create a global variable associated with that function. While it can only be referenced from the function in which it is declared, its value will be preserved across calls to the function.[3]

Now it's time for our first code example. Use your newly acquired Unix mastery to navigate to the code examples, into the `CH1` directory,

[3] What if you declare a global variable as `static`? It actually has very little to do with the local variable `static`. You'll find out in 4.1.

and open up global.c in your preferred text editor. Since this is your first time looking at C code, there's a few things you should note. First of all, the function main is a special function that is called automatically when the OS runs the program (as you might guess). Also, 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 explained in detail in section 4.2. I tried to avoid putting this kind of "please ignore for now" stuff in this text, but it's unavoidable in this case. You might also ask "What's the void keyword doing as the 'argument' to the functions with no arguments, including main?" 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 code that tries to call with arguments will not compile, which is what we want.

Now have a look at the three functions, and main. What do you think will be printed? Compile the code by running make global and run it by doing ./global.o. It's now crucial you have make and gcc installed.[4]. Did it print what you expected?

1.6 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's a pointer?

Memory in operating systems is byte-addressed. That is, each byte of memory available to the system is labelled with a number, called its memory address, 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.

[4] for more information about make see the appendix

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, like `int *x;`. 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. The idea is that it's a kind of mnemonic. [5] For example, what is this code doing?

```
int *x1, x2;
```

It declares a pointer to `int`, `x1`, and an `int` `x2`. The idea is that `x2` is already an int. But if you want an `int` from `x1`, you need to dereference it. This is also why we put the asterisk where we do. If you wrote

```
int* x1, x2;
```

You might think you were declaring two pointers, when in reality, only `x1` is a pointer.

By the way, have you ever seen hexadecimal numbers before? The idea is that the most natural way to write down numbers that live inside a computer is in base two, otherwise known as binary, since that's how they're represented in the machine. The problem is that it's not exactly compact. A 32-bit number takes, well, 32 digits to write down. But if you use a larger base, it'll be more compact. So why not decimal? We all like decimal. The problem is a decimal digit does not directly encode any number of binary digits. For that, we'll need a base that's a power of two. Base-16, or hexadecimal is a happy medium. One hex digit encode four binary digits (get it? $2^4 = 16$), so a 32 bit integer can be represented with only 8 digits. The problem is that there are only ten arabic numerals, but we'd need 16 for hexadecimal. So the digits 0-9 mean what they usually do, then we use the letters `a` through `f` for the numbers 10 through 15. You can write hexadecimal integer literals into your C code by starting them with 0x, as you'll see below.

Now back to our regularly scheduled discussion of pointers. Here's an example of how to work with pointers

```c
int x=0xdeadbeef;
// get a pointer to x
int *xptr=&x;
// read the value being pointed to using a dereference
int y=*xptr;
// mutate the value being pointed to.
*xptr=0xb00b1e5;
```

If one executed the following code in some function, `x` would be equal to 0xb00b135, and `y` would be equal to 0xdeadbeef. The latter might seem strange. What the dereference operation that defines `y` is doing is **copying** the data pointed to by `xptr` into the variable `y`, which is not mutated after that. ***The syntax of pointers is going to pervade this text. Get used to it.*** Thankfully, there's a code example coming up.

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.

So now let's think back to the `mutate_int` function from before. How could we accomplish what we set out to do? Using pointers, it's pretty straightforward. If you wanted to set any integer to 5, you would need a pointer to that integer, not the integer itself, like so

```c
void mutate_int(int *xptr)
{
  // set the data pointed to by xptr to 5
  *xptr=5;
}
```

Navigate to the code examples, and go into the CH1 directory. Open up `noop.c`. You should recognize both of these functions. Compile with `make noop`. Did you notice the compiler warning you get from our flawed mutate 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 the code with `./noop.o`

1.7 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 to integers? Not exactly. It actually increments the pointer by as many bytes as the size of the 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:

Line 4 of this block is your first exposure to typecasting in C. Typecasting does different things at different times, but when you cast one pointer type to another, you are essentially asking the compiler to keep the numerical 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 it will print? The second function is the code from above ... sort of. [5]

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.

A final note. Pointers are just numbers. That means any operator you can use on a number, like equality, comparisons, or other arithmetic operations will work as expected for a number. Different operations may or may not make sense, but they will work as expected.

1.8 Hanging pointers

Earlier we talked about the scope of local 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.

When a function is done executing, you can't predict what the values of the local variables will be. Just bear in mind that as far as the language is concerned, the behavior is undefined, so just don't do it.

[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.

1.9 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 undefined. [4] Extra extra extra credit if you can guess why the returned value is what it is... (hint: what does `printf` return?)

2. Open up `ex1.c` (ignore the name), and have a look around. Read the comments. [1] Do a `make ex1`. You get a bunch of warnings, right? What does it do when you do `./ex1.o`. Fix the `swapnum` function so that it actually swaps the values of two numbers.

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

2.1 malloc and free

All of the variables we've been working with up to this point have had either automatic or static storage duration. That is, their lifetime is either the duration of the program, or when the variable is in scope. It turns out that there is a third kind of storage: allocated.

The idea of allocated storage is that you ask the system for a certain number of bytes of memory, and the system goes out and finds that many contiguous bytes, and gives you a pointer to it. That data's lifetime is as long as you need it to be, and there are no scope rules or anything like that. That said, sometime before your program exits, you need to tell the system that you're done with that memory. The C standard library has functions for both of these processes: `malloc` and `free.`

Before we go into it, I should talk a bit about how memory works in real implementations. 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 programs must always be following this paradigm, at least behind the scenes. So how do we get the kinds of storage we've described. Automatic storage is almost always implemented using an execution stack. A program is given a certain amount of memory at startup to keep its stack. Each function has a "stack frame" which has storage for all of that function's local data, and the stack frames pile up with each successive function call. Then, as each function call exits, the stack frame is popped, freeing the memory to be used for other things. When the program exits, all the stack memory is automatically returned.

Allocated storage, on the other hand, is usually implemented with a heap. The heap is a part of the memory that the OS manages At runtime, your program asks the OS for a certain amount of memory, and the OS goes out and finds it. When you're done with it, you have to tell the OS that that memory is no longer needed, or the OS will never know that memory is free to be used for other things.

I'm just gonna tell you right now, that I'm going to get sloppy with my terminology in the rest of this book, referring to variables as "stack-allocated" and "heap-allocated." For the rest of this text, interpret "stack-allocated" to mean "automatic storage" and "heap allocated" as "allocated storage."

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, as we discussed in the section on data types. 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. There is no way to determine from just a pointer how many valid bytes follow that location, you have to keep track yourself.

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

Also, `malloc` can return `NULL`[1] if it fails, so technically you should be checking for that after every single call to `malloc`. But in practice, if `malloc` fails, it means there's big trouble, and your program is probably screwed, so most people don't do it.

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 undefined. 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 kernel 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.

2.2 arrays

About fucking time. I mean, seriously, this is like page 17 or something. Why did it take this long? Well, in order to understand C arrays, you really need to understand pointers, because they're extremely close concepts.

The way you allocate an array of three integers in C is to do the following:

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

This allocates three integers' worth of contiguous automatically-managed space with the type "array of `int`." You can read and write to arrays like so:

[1] `NULL` in C is just 0 cast to a pointer. Don't dereference it.

```c
int arr[3];
arr[1]=2;
printf("%d\n",arr[1]); //prints '2'
```

You can also initialize arrays at the same time you declare them.

```c
int arr[]={1,2,3};
printf("%d\n",arr[1]); //prints '2'
```

The compiler can figure out the size based on how many elements you give.

But I said that pointers and arrays are very similar. That's because everything you could do with arrays could be done with pointers. Say you did the following

```c
int arr[3];
int *ptr;
ptr=&(arr[0]);
```

The variable `ptr` would hold the memory address of the first element of `arr`. From there, you could read and write to `arr` using `ptr`.

```c
int arr[3];
int *ptr;
ptr=&(arr[0]);
*(ptr+1)=2;
printf("%d\n",*(ptr+1)); //prints '2'
```

In fact, to make this easier, you can still use the square bracket notation. For a pointer `ptr`, `ptr[i]` gets translated by the compiler to `*(ptr+i)`. They are completely equivalent in every way.[2] Also, if you refer to the name of an array, it gets translated to the memory address of its first element. So this code works.

```c
int arr[3];
int *ptr;

ptr=arr;
ptr[1]=2;
printf("%d\n",ptr[1]); //prints '2'
```

[2] Fun fact: some C compilers interpret the above equivalence very, very literally. So if you wanted to access the third element of `ptr` you could either do `ptr[3]` or `3[ptr]`. Since addition is commutative, it really doesn't matter, though it's some of the worst coding style concievably possible.

So...what the hell is the difference between an array and a pointer? Not a whole lot actually. One difference is that the array is not assignable. So you can't have

```
int arr[3];
int arr2[3];
arr=arr2; // NOT ALLOWED
```

Whereas pointers can absolutely be assigned to. Another difference is that `sizeof(arr)` would return three times the size of `int` on your machine (so 12 on my setup), and `sizeof(ptr)` will just return the size of a pointer, which in my case is 8. This is a very, very subtle distinction, and a major source of bugs.

All of the arrays we've been talking about have been using stack storage. These are nice, since they're automatically managed, but they're tied down by scope. You can't, for example, return an array from a function. Have a look at the following code.

```
int *mkarr(void)
{
  int arr[]={1,2,3};
  return arr; // NOPE
}
```

The types work out fine, since referring to the name of an array gives you the pointer to the beginning. So what's the problem here? That's right: you return dangling pointer. By the time the function exits, the local variable `arr` is out of scope, and referencing pointers to it causes undefined behavior.

This is why we often allocate buffers on the heap.

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

This `buf` can be used in every way exactly how you would use an array, except the memory is available for use until you `free` it.

```
int *makearray(int n)
{
  int *arr=malloc(n*sizeof(int));
  for(int i=0; i<n; i++)
        arr[i]=i;
  return arr.
}
```

is totally fine. 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. In large-scale programs, it's a lot to keep track of. The other disadvantage is that `malloc` is (sort of) 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.[3]

Technically, it's up to you how you want to allocate. However, the limitations of stack-allocation is usually a dealbreaker, so you better get used to `malloc`.

That was a hell of a build-up to just index an array, huh? But now you understand it top to bottom. 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).

Arrays as function arguments

I bring this up because the syntax is, in my opinion, very unclear. Say you want to have a function that takes an array as its argument. One thing that makes sense is to take a pointer to the beginning of it, and work with it that way. Of course, you would need to keep track of its size, since you can only see the beginning. That would look like

```c
void take_array(int *buf, unsigned num_elem)
{
  for(unsigned i=0; i<num_elem; i++) {
    // do things
  }
}

int main(void)
{
  int buf[]={1,2,3};
  take_array(buf, 3);
}
```

Makes sense, right? But C offers another syntax for taking an array. You can have a function that looks like

[3] In performance-critical applications, when a buffer 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.

```
void take_array(int buf[])
{
  //do things
}
```

You might think that the prescence of the square brackets preserves the “arrayness” of the input. You might think you could use the `sizeof` operator to get the size of this array, and therefore use `sizeof(arr)/sizeof(int)` to determine the number of elements in the array. **This is not the case.** Putting `int buf[]` in the type signature is **completely equivalent, in every way** to `int *buf`. In either case `buf` is a pointer to the beginning of the array, and there's no inherent way to determine the length of the array, so you'll need to keep track of it. The only reason to use one or the other is because you feel that perhaps one or the other might be more clear to one reading your code. It's up to you.

Open up `arr.c` in the chapter 2 examples. What do you think it will print? Type `make arr && ./arr.o` to compile and run. Did it print what you expected?

2.3 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;
```

This saves you from having to keep track of the string's length. When you're iterating over it, you just have to iterate character by character, until you get the null character, so to iterate over a string, you could do

```
char c=*str;
while (c!=0)
{
  //do things
  c=*(++str);
}
```

The while condition could also have `while (!c)` instead. Think about it. Remember what's in the guards of conditionals. It's how you'll often see iteration over strings done in the wild.

Be warned. Null terminated strings are one of the biggest sources of bugs in the history of computer science. I'm not even kidding. Always be careful when working with strings.

Other than null termination, 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`.

What if you wanted to initialize a string on the heap? You `malloc` sufficient space for it, and copy it in. The standard library provides `strncpy` for copying strings around. Read the docs for it.

There's one last thing to note. I'm sure you've had enough of the whole array vs pointer thing, but we have to come back to it. There's two ways to initialize a string based on a string literal. In case you're not up on the lingo, a string literal is the stuff you write between the double quotes, like "abcd" above. Here are the two ways

```
char str1[]="abcd";
char *str2="abcd";
```

Usually, string literals in C are stored by the compiler in some read-only segment of your executable. Attempting to write to a read-only segment is a sure-fire way to generate a segmentation fault. So with that said, what's the difference between `str1` and `str2`? `str1` is an array, with the string literal **copied** into it. This is 100% your memory to work with, and you can read it, write it, do whatever you want with it, it just happens to mirror the string literal. `str2` on the other hand, stores a pointer to that read-only memory I was talking about earlier. Read it all you want, but from a language perspective, attempting to mutate it will result in undefined behavior. On most machines, this will usually result in a segmentation fault, and the tragic death of your program.

2.4 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. *These next exercises are the first real coding exercises in the text. You have the text, my examples, and the whole internet at your disposal. Good luck.*

   Write a function in `stringstuff.c` 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 as arguments, 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`. Some of these are implemented in the standard libraries, but they're pretty straightforward to code

6. In the file `myecho.c`, reimplement the Unix utility `echo`. First, figure out what `echo` does (try typing `man echo` at the console). Then figure out how to take arguments on the command line (`void` is not the only type signature you can use to declare `main`). Then implement. And don't worry about parsing options like `-n` or `-e`. There are libraries to help with that (it's actually fairly nontrivial to do it yourself). So for now just implement plain `echo`.

   By the way, my solution for this is about 9 lines, total, so if you're writing much more than that, you might not have the right idea.

> Keep in mind that there are standard library functions to print strings (I've used `printf`'s `%s` format in my examples), so you shouldn't need to iterate over characters.

Make sure your solutions have no compiler warnings, and if you use `malloc`, use `valgrind` to test for memory leaks. The next chapter's exercises will give you much more practice with arrays.

Chapter 3

Structs

3.1 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 very often use vectors in three dimensions. 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 to use, so good design is crucial to your team's success. The first probelem, obviously, is how to represent a vector. 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.

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 there's three or more vectors floating around, 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 the fact that arrays are just pointers (i.e. their size is not part of their type) 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?

3.2 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 variables 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 abstractions like abstract classes and interfaces, since none of these 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

[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).

```c
typedef float real_t;

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

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." Then we can declare fields with that type just like we would 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?

```c
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." This is usually the syntax you'll usually use to define datastructs in C.

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

```c
vec3d v;
```

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

```c
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. The above code is first allocating a buffer in heap memory, and mutating it. Then, the dereference operation fetches that memory from the heap, and copies it into a stack-allocated variable, which is then itself mutated, leaving heap memory unchanged. This is just like the first example of pointers I gave in chapter 1. If you use the dereference operator on the right side of an assignment, it copies the contents being pointed to, resulting in a completely independent piece of data.

By the way, an important 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.

There's a couple of gotchas too keep in mind when passing structs into functions. Have a look at this code. What does this function do?

```
void mutate_struct(vec3d v)
{
  v.x=2.0;
}
```

Nothing at all. This is the struct analog of the `mutate_int` function from chapter 1. When you pass a `vec3d` into this function, it gets copied into a variable local to the `mutate_struct` function, and the original struct stays the same. To actually mutate a struct, you'd have to do this.

```
void mutate_struct(vec3d *v)
{
  v->x=2.0;
}
```

This way, instead of the whole struct getting copied into a local variable, the pointer to that location gets copied, so both the local variable and the original pointer refer to the same memory location. That way, when you use the -> syntax to mutate the struct, you're referring to the same struct you thought you were referring to.

To get a more concrete idea of this, have a look at the following code

```
void mutate_struct(vec3d *v)
{
  v->x=2.0;
}

int main(void)
{
  vec3d *v=malloc(sizeof(vec3d));
  mutate_struct(v);
  printf("v.x=%f\n", v.x);
  return 0;
}
```

Think about what this did.

1. We declare a variable of type pointer to `vec3d`, and initalize it to be a pointer to a segment of memory of the right size.

2. We call `mutate_struct` on that variable, copying the numerical value of the pointer `v` into another variable of the same name, local to that function. At this point, the two `v`'s point to the same heap-allocated data.

3. That function mutates the data in heap space. The function call exits and `mutate_struct`'s `v` variable falls out of scope.

4. Since the `v` that lives in `main` is still pointing to that same place in heap memory, the value printed is the one that was set in `mutate_struct`, 2.5.

A better book would have a nice diagram of stack and heap space at this point, with the values of the different variables diagrammed out all nicely, but I'm not capable of making those. Try and understand the steps I wrote above.

3.3 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. Not as if that would be a crucial step in doing a major software test in a student satellite project or anything like that.

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, with 32 bit `ints` and 8 bit `chars`, and you have the following struct

```c
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 live 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 about trying to cast arrays of bytes to structs. So the padding could be before or after c. What the compiler will never do is change the order of the fields. It just doesn't make that choice.

Now back to our vector library. Head over to the examples for this chapter and 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...

3.4 OOP(ish) in C

One thing you might have noticed about that vector library. The functions all have 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 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.

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

Now check out `minheap.c`. Guess what it implements. For a usage example, I implemented 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.

3.5 Exercises

This is where the shit hits the fan. 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.

Some advice: you will likely find debugging C code is significantly harder than debugging high-level languages. More than ever, you'll be well-served by thinking deeply about your programs **before** you write them (which you should be doing in high-level languages anyway). Seriously, think twice about what you're doing. When you index an array, **prove** to yourself that you can't go out of bounds. That said, when you have bugs, staring at the code and `printf` are obviously your first line of defense, but if those fail, refer to the appendix section on debugging. 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`. See how the time the sort takes changes. Remember how I implemented the buffer doubling when you're doing the later excercises. If you screw up big time and can't undo your changes, you can do

   ```
   git checkout master -- minheap.c
   ```

   to get back to my version. **Note that this will permanently erase your changes.**

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

On a side note, this will probably only require you to change five or so lines of code, but will force you to figure out all of my code. That's the point of this exercise. That kind of thing is a big part of being a programmer. And either way, figuing out how I did the min heap is probably for the best, since the real stuff is coming in the next exercise.

2. The file `arraylist.c` includes a struct definition and some function declarations, definining a type with behavior similar to Java's `ArrayList`. Implement those functions. Use as many helper functions as you think appropriate.

   Note that your arraylist is going to store elements of type `void*`. Since any pointer type can be cast back and forth to `void*` easily, this means your arraylist can store any pointer. It's as close as you're gonna get to polymorphism without doing some more complex stuff like tagged unions (which I might talk about later).

   Write functions to unit test your data structure, and run them in `main`. Make sure you have tests that cause the arraylist's size to double. Also, make sure to run the tests in `valgrind` or a similar tool to check for memory leaks.

3. Training wheels are coming off. Use your `arraylist` implementation to implement a hashtable with chaining. If you don't know what that is, check wikipedia. Have it map from `void*` to `void*`. Use whatever hash function you want, just find a decent one. Also, just hash on the numerical value of the `void*` key – you don't have to support more sophisticated notions of equality.[4]

   I provide no boilerplate code, nothing in the `Makefile`. Figure it out. Note you will probably need the same `include` lines as in `arraylist.c`. We're finally going to be explaining those in the next chapter.

   Include your arraylist by just copying and pasting the contents of `arraylist.c` (except for the `main` function) into your new file, and proceeding from there. I promise we're going to learn to organize code starting next chapter. Some advice: first implement a hashtable with a fixed number of buckets. When you have that working, then implement rehashing/resizing.

[4] We'll be getting to this in section 6.1

As a mild hint, there's two ways you could do this. One way would be to have two arrays in your hashtable, one for the keys and one for the values. Or, you could define a `struct` that represents a key and value, and have your `arraylist` buckets hold pointers to these structs. They should have the same time and space performace, so it's really up to you which you think is a better design.

Make sure you hashtable can do the following. Name the functions whatever seems reasonable to you.

- add a key value pair, and update the value if the same key is added more than once
- get a value based on a key
- test if a key is in the table
- remove a kv pair

Again, make sure to use `valgrind` to test for memory leaks.

I could probably put a ton more stuff here, but you should probably wait until you get the next couple of chapters under your belt before you start building bigger stuff.

Chapter 4

The Compiler & You

This chapter is under active construction. This stuff is very, very hard to explain to someone that doesn't have much C experience, so this will probably be rearranged multiple times.

The code you've been writing up to this point has been very straightforward, single-file programs. Obviously, C programs at scale are spread accross dozens or hundreds of source files. Even small programs are organized into a few files. At this point, the easy thing to do for me would be to say a few basic things about how to get your multi-file program to compile, and you'd be good.

- Put your struct definitions and function declarations into a "header file" with a `.h` extension.
- `#include` that header file in both the file that implements those declarations, and the file that will eventually call those functions and use those structs.
- Make sure only one source file has a `main`.
- Make sure `gcc` has access to all your source files. If you're following the style of my `Makefiles`, make sure all your `.c` files are listed in the `SRC_<WHATEVER>` variable.

Those are the things that I figured out in trying to get my C projects to compile, and if you keep them in mind, you'll manage to figure out how small projects are compiled. To be honest, you could probably do all of the exercises at the end of the chapter with only that knowledge.

But the fact is that, especially at scale, one of things that sets coding in C apart from high-level languages is that in order to write good C code, you have to have much more awareness of what the compiler is doing behind the scenes.

It turns out that `gcc` has been doing a whole lot behind the scenes to compile your programs, despite only being invoked with a single command. The compilation of a C program has three major stages: preprocessing, compilation, and linking, all of which are independent. The GNU compilation system is backed by what's called a compiler driver system that organizes these steps for you. Before we talk in detail about these steps, there's a few pieces of overall information you should keep in mind.

First of all, before the linking phase, each source file is its own separate entity. The preprocessor and compiler operate on each file individually. There's a very good reason for this. Mozilla Firefox is written primarily in C, and I've heard it used to take upwards of 20 minutes to compile from scratch. If you just edit one file, then it'd be ideal to only have to recompile that file, and then only have to run the linker.

Speaking of which, even though the linker is the final stage of compilation, I feel it's the most natural place to start.

4.1 The Linker

Before I say anything else, let me say that I'm not planning on explaining how linker works. Your best bet for really learning this is to either read a fully-fledged textbook, or to take a course on systems programming and/or operating systems. That way you can get a full context on a lot of the things I'm talking about here. That said, what I will explain is the basics of what you need to know to make projects with multiple files in C.

So we understand that each source file you provide is compiled individually. As the compiler is going through a given C file, its job is to first of all, do things like compile the code of all the functions into machine code. But one of its jobs is to organize the data in that file into what's called a symbol table, which will eventually be a part of the intermediate binary that gets created from that one C file. The binary that gets generated from the individual C files is called a relocatable object file. [2] When the linker has been through and connected everything and found a `main` function to run on startup, the final output is called an executable object file. I'm not sure why they call them objects, but it's the reason for the .o extension. It categorizes the symbols (i.e. variables) in that module (i.e. file). The most relevant category is that of the global symbols. These are the global variables and functions defined in the module. These symbols are meant to be visable to other modules. Then there are the local symbols, those symbols only referenced within the module in question, and are not visible to the linker. But there's one more category.

Obviously, if these files are supposed to produce one coherent program, these files are going to reference one another. The last category is that of the external symbols. These are symbols that live in some other module. **This is where we can reference other files**. How does the compiler know to create these references?

One way is to use the `extern` keyword. Declaring a variable with the `extern` keyword declares the nature of the data to the compiler, but it does not allocate storage for that variable. Therefore, it relies on that variable being defined someplace else. If another module has a variable of the right name and type, the linker will find and connect those variables.

Head to the chapter 4 examples. You'll notice now there's a bunch of folders. Now that we're doing multiple files, it'd get too chaotic to have everything in one directory. Go into the first example. The first file I want you to look at is actually the `Makefile`. See how I give both source files to `make`? Now look at the two C files in the directory. One file references a function defined in the other. And that file references global data defined in the first one. Compile with `make` and run with `./main.o`.

Makes sense right? It turns out that we were being redundant by declaring the `printstuff` function as `extern`. Try deleting the `extern` keyword from the function declaration. **Recompile** and rerun. Still works, right? It turns out `extern` is implied in function declarations, so don't bother writing it. Now we're gonna get really crazy. Try commenting out the whole declaration of `printstuff` in `main.c`. Recompile. You get a warning right? But it runs just fine. When you attempt to call a function that has been neither declared nor defined, it's called an implicit declaration. It turns out that certain C standards allow implicit declarations, and can use them for linking. It's always bad style, though, so don't do it. I just thought you should know what happens if you don't have a declaration. Bear in mind that the external reference to the `buf` variable had no redundant stuff. Try messing with any of it, it won't work as expected.

So now we know that we can use `extern` to use things from other files. But does that mean that anyone linking against your code can reference any of your functions and global variables, and you don't have anything you can say about it? Remember way back at the beginning, when I talked about how `static` local variables, I said that the `static` keyword has a completely different meaning for global data? Here we are. If you declare a global variable or function with the `static` keyword, it makes that data invisible to the linker. This is actually very much like declaring data as `private` in Java or C++. We'll talk more about that in the section on code style.

Now, if you're a design-minded person, you should be worried sick about the way a lot of this is done. With what you currently know, in order to reference variables and functions from other files, you need to type in (or worse, copy and paste) all the function and variable declarations you want to use. That's a big issue. Well, remember the whole 'preprocessor' part at the beginning of compilation? Let's chat about that.

4.2 The Preprocessor

The preprocessor's job is to prepare the code you wrote to get read by the compiler. That means doing things like rearranging the whitespace in the file into some regular format that the compiler knows how to read. Its main job, though, is to interpret preprocessor directives. Preprocessor directives are a bit like a mini programming language that lives within your C code. But its purpose, rather than general computing, is to manipulate the text of your code in various ways. That will make more sense as we explore some of the things it can do.

First a bit of general information. You can recognize a preprocessor directive by the fact that they begin with a # and don't end with a semicolon. In order for the directive to be properly recognized, the # character must be the first non-whitespace character on the line. Also, the preprocessor uses the newline character as the end of a statement, kind of like a semicolon in normal C code. So if you want to have a preprocessor directive extend over multiple lines, you have to "escape" the newline using a forward slash, so it would look like

```
#define ONE_MILLION \
  1000000
```

Now let's move on to the applications. Note that this will not by any means be a complete description of the preprocessor, only of the most common uses.

### include

This one you've seen in every example file so far, and you probably have some speculation about what it does. If you're like me when I was learning C, you probably think it's something akin to `import` in Java or Python. In reality, it's nothing anywhere near that complicated. `#include "file"` simply copies and pastes the full text of that file into your code at that location. I'm not kidding, it really is that simple. The syntax `#include <file>` is completely equivalent. The only difference is that the angle brackets are stylistically only used for including standard library files. So let's think about a few possible uses of this.

There's a lot of stuff, even that you were coding in the last chapter's exercises, had a lot of stuff that you might well want to move to another file. Things like `struct` definitions, functions declarations, and typedefs aren't really things you need to have right in front of you when you're implementing some algorithm. The common thing to do is to move this kind of data into a separate file called a "header file" (usually with a `.h` extension).

On the subject of the `include` directive, it doesn't always have to be used for code organization. Say you want to initialize an array. So you do

```
int arr[]={1,2,3,4,5};
```

That's fine. However, in some engineering applications, it's necessary to have these arrays be many thousands of elements long. It would be insane to hard-code these constants in the source code. But what if the values could be in a comma-separated list in a file, and then `include`d? It would look just like this:

```
int arr[]={
  #include "params.txt"
};
```

Where `params.txt` is simply a comma separated list of integers. The preprocessor copies and pastes that text right in between the curly braces, and it'll work just like if you wrote them all into the source code.

That said, try to avoid little tricks like that unless it's really necessary. It's pretty easy to write some pretty horrific code doing stuff like that, but just know that the capability is there if you need it.

define

Code always has little constanstants that come up all over the place. It might be math constants, sizes of things, strings, anything. And it's usually bad style to just hard code them, for a ton of reasons. It's easier to change, it's easier to remember a word then a number, you should know the drill by now. But for now, assume you want to have a constant. One option would be to have some global variable like

```
const int FIVE=5;
```

And that's fine. But the problem is that the compiler will very often choose to reserve memory for that variable (since what if someone wants to link against it, and stuff like that), so your program will have to read from global memory to use this variable, which could evict the cache, slow things down. But then again, how does your program work with constants when you hard-code them? It turns out most processor instruction sets have instructions that take an "immediate value"–a number that's actually hard-coded into the instruction. So there's no memory read involved. But I thought the point was not to hard code? Say you did

```
#define FIVE 5
```

The preprocessor now knows that every time it sees the string `FIVE` in your code, it should replace it with the number 5. Since the preprocessor is really only doing string manipulation, there's a lot of flexibility in what you can define.

```
#define SQRT2 1.414213562
#define NAME "Dan"
```

It turns out you can also do this

```
#define VAR
```

And `VAR` will be defined to be nothing. There's some interesting stuff you can do with the preprocessor that we'll come back to later.

The preprocessor can do more than define variables, though. It can also define mini functions called macros. It looks like this.

```
#define SQUARE(x) x*x
```

This is still just a copy and paste operation. When the preprocessor sees `SQUARE(5)`, it just plugs five in for x in the macro definition, and pastes all of it in where you had the macro call. Of course, it doesn't have to be a literal, it could be a variable name, a function call, an expression...

But you have to be careful. The copy-paste happens before any evaluation. So say you did this.

```
SQUARE(1+5)
```

You probably wanted this to evaluate to 36. Well let's think about that. When the preprocessor expands, this, it'll look like this

```
1+5*1+5
```

The compiler knows all about PEMDAS, so this evaluates to 11, not 36. We could have fixed this with parentheses, so you just need to understand what the preprocessor is doing, and be careful when you make macros.

But why do this at all? What's wrong with normal functions? There's a couple of advantages. One is that there's overhead to calling functions in C. In the machine code, you have to save a bunch of registers, screw around with frame pointers and stuff, then finally jump over to the function code. It's not a huge deal, but if you call a function a lot, like in nested loops, it adds up. Since the preprocessor just copies and pastes the function body in, there's none of that. Of course, C also provides the `inline` keyword, which we may or may not talk about later, to do this, so it's really not a big deal, but there it is.

Another cool thing is that there's an interesting kind of polymorphism to preprocessor macros. Take this macro for example.

```
#define INCR(x) x++
```

This macro will work on any data that can work with the postfix ++ operator, `int, float, void*` pretty much anything, which you can't do with a normal function. The last interesting feature I'll talk about is this. You can actually get the name of a variable in code in a macro. Say you had a bunch of arrays of three integers, and you wanted to print their contents, but you needed to keep track of them. You could define this macro.

```
#define PRINT_THREE(arr) \
        printf("%s={%d,%d,%d}\n", \
        #arr, \
        arr[0], arr[1], arr[2]);
```

That thing with the pound sign gets the name of the variable you pass in as a string literal, so if you had an array x with the contents, 1,2,3, PRINT_THREE(x) would print x={1,2,3}. Useful for debugging.

4.3 Exercises

I really couldn't think of any exercises for this stuff. The next chapter is really going to just be applications of stuff in this one, so we'll wait until then.

Chapter 5

Code Organization & Style

All this time, we've been coding and talking about coding in a very pragmatic way. We've had such a small subset of C to work with, we haven't really had much of chance to talk about any thing other than how to write correct code, and style has taken a major back seat. In the real world, however, code style can make or break a project. You've probably learned about various principles of code design in past CS classes, probably in the context of some kind of object-oriented language. The purpose of this chapter is to explain how we can apply these principles in C.

5.1 Encapsulation

Java programmers will be very familliar with this. The idea of encapsulation is essentially that there should be a solid brick wall between the interface to the code, and its implementation. After I described how `#include` works, you might have wondered why you would use the linker. Why not just include all your other implementation files, and just compile the whole program at once. Besides the fact that that would make for extremely long compilation times, it would make encapsulation impossible.

The interface of some piece of code is the bare minimum someone needs to know in order to use the code. In the case of a C program, the user will usually want to call functions. Well, in order to use a function, a user would need to know its name and type signature. A specification describing what the function does would also be nice. A user might also want to know the definitions of various structs, if they're user-facing. Finally, if the user needs to link against any global variables, they'll need to know their names and types.

In C, this information comes in the form of function declarations, struct declarations, and external variable declarations, and is usually moved into a file separate from the implementation called a "header file." The implementation file then `#includes` this file, to make sure the right interface is actually being implemented. Then, the caller code also includes the header file so that it has all the same things declared. Then the caller simply has to link against the object code of the implementation, and everything will link up fine. The caller doesn't even need to have access to the implementation file, allowing the implementer to hide his or her source code.

An aside: Header Guards

The C preprocessor can do more than `include` and `define`. It turns out that it has a kind of conditional structure that you can use to do all kinds of tricks. In general, too much preprocessor wizardry is a recepie for some really weird-looking, obfuscated code, but there's one little trick you'll see a lot.

What if you accidentally `#include` a header file multiple times? The file will be copied and pasted twice, and C does not allow multiple declarations of variables and functions, so you'll get a bunch of really weird-looking compiler errors complaining about "multiple declarations" of all your functions.

You could just say "well, just don't include your files multiple times," and that's fine, but in large projects that's sometimes difficult. What if your header file includes a header file that includes a header file that ... that includes one of the headers you included? You'll get the aforementioned multiple declaration errors and have no idea why. That's why you'll often see code like the following at the beginning and end of header files.

```
#ifndef __FILE_H_INCLUDED
#define __FILE_H_INCLUDED

/*
  code goes here
*/

#endif
```

So what's going on here? This is the preprocessor's version of control flow. The first directive tests if the variable `__FILE_H_INCLUDED` is defined in its set of defined variables. If it's not defined, the code between that and the `#endif` is included, and therefore the variable is defined for the remainder of preprocessing. If it is defined, without regard to what it's defined as (or whether it's defined to be anything in particular), the preprocessor will skip to the `#endif`, **without any of the code in between making it to the final product.**

The end product of this logic is that if you include a header more than once, the `#ifndef` will fail, and the header file will effectively only be included once, keeping you safe from the multiple declaration bugs. Like I said, it's something you see pretty ubiquitously in production C and C++ code.

Just make absolutely sure the name of the variable you define is unique. You can imagine what kinds of issues that would cause.

Abstract Data Types

There's another point I want to make about encapsulation. One of the many things that differentiates C from object-oriented languages is that there is no built-in notion of `public` or `private`. Sure, you can hide data from the linker using the `static` keyword, but it doesn't help with datatypes. Very often when defining some datatype backed by a `struct`, the data in that struct is crucial to the representation of that datatype.

There are many examples of this, even in the code examples we've seen so far. For example, take the `vec3d` type from the beginning of chapter 3. When I wrote this type, I intended it to be *immutable* data.[1] That is, none of the operations I encoded into functions alter the data in the vector. But if the struct backing that type is exposed to the user, there's nothing stopping the user from disregarding that intention and just writing `v->x=2.0`.

But that's not terribly nefarious. A bigger issue can be found in the `minheap` I wrote in that same chapter. Fields like `cap` and `size` are crucial to the functionality of that datatype. If `size` is screwed up, the code will think it has more nodes than it actually does, and will perhaps read garbage values, values that have already been popped, or it might forget about valid values. If `cap` is screwed up, it'll forget about the size of its own buffer, and we might overflow it, leading to unspecified behavior. So if a user of that code is under the accidental impression that

[1] Immutability is a hallmark of functional programming languages. My love of functional programming languages is going to become more and more apparent as we go on from here.

it is his or her responsibility to mutate those fields, it will cause the code to malfunction in strange ways.

In CS courses, it is usually good enough to just say, "we can just write a comment saying not to mutate those fields, and any mistakes will be on the user." But in a company's production code, or engineering applications, or any other serious purpose *that isn't good enough.* It is your responsibility to make sure it is as hard as possible for your clients to misuse your code. So we want a way to hide the details of a datatype from a user. Now, navigate into the examples for this chapter, and into the `heap` directory. I first want you to open up `minheap.h`. The first normal line is

```c
typedef struct heap *minheap;
```

This should look a tad strange, especially when you consider that calling code, which has no notion of `struct heap *`, is meant to include this. It turns out C does allow this. Unless it is later defined, the compiler interprets `struct heap *` as an "incomplete type," kind of like `void*`. The point is that users can only really see the type `minheap`, and they only know it's a pointer to something. Keep in mind that even though `minheap` doesn't look like a pointer, it actually is, so you get all the usual benefits. They can't create them or edit them without going through the functions I define. Now if you look at the `minheap.c` file, you'll notice I use the `minheap` type throughout, but it can be dereferenced since that file has a definition for `struct heap`, so it's no longer an incomplete type. Also note that I used the `static` keyword to hide all of my helper functions from the linker. Finally, in `main.c`, all we have to do is include the header, and we're good to go. Make sure to look at the `Makefile` as well. Note in this case, we use both source files, but in principle `main.c` could have just been linked against a relocatable object compiled from `minheap.c`.

Isn't that so much nicer than having it all in one file? Compile with `make` and run with `./main.o`. By the way, since the `main` I wrote records the time of the `heapsort` operation, this is a neat time to play around with compiler optimizations (look at the appendix). Crank up `num_elems` a bunch higher and see how much better the performance is between no optimizations and `-O3`.

5.2 Input via Extern

Though I consider this to be very strange and not the best code style, it's a design paradigm that seems to be common enough to be worth mentioning. Also, it's more examples of `extern`. This is something that you see a lot in embedded systems coding.

What if, instead of having function calls that pass in the inputs to an algorithm as arguments, you have a global struct that takes in the inputs, and then a `void` function that runs the algorithm, and copies the outputs into another struct? Have a quick look at the `quadratic` directory and piece together what's going on. Note that all the struct types are exposed here, since users of the code *must* know what the structs contain in order to use the code. It is then the algorithm implementer's responsibility to handle bad user input.

Look around the project. I wrote some comments in `quad.c`. Note how I handled all the potential improper input cases. Compile and run the same way as the heap.

5.3 Exercises

I'm not going to do numbering, since it's really one exercise. By the way, for this exercise, don't worry so much about abstracting all the types for arraylists, hashtables, btrees, etc. The set is abstracted, and that's the only one we're considering to be "client-facing." Go into the `set` directory. You'll notice there's a header file in this directory. This header defines an interface for a set abstraction. There are two subdirectories, `hashset` and `treeset`. The former is empty, save for a `Makefile`. In this folder, implement a set using your `hashtable` implementation from chapter three. Here's the plan:

- Refactor your hashtable implementation using good code design. Begin by removing your arraylist implementation and putting it into a header file and implementation file.

- Move the hashtable interface into a header file.

- Finally, make a `.c` file that implements the set interface using that hashtable. Just have a hashtable that maps all of its keys to `NULL`. Make sure you have a type named `struct set_struct` somewhere in your code, even if it's just a raw `typedef` of your hashtable type.

- Make a `main.c` to put your tests into.

All in all there should be four C files in the directory. `arraylist.c`, `hashtable.c`, `hashset.c`, and `main.c`. The `Makefile` assumes they have those names, so you'll need to change it if you want to name them differently. When you're ready, compile with `make` and run with `./main.o`.

Now have a look at the `treeset` directory. This time, I put a header file in with the `Makefile`. This shouldn't be too hard. First, just implement a binary search tree. I put some stub code in the header file. Feel free to write recursive functions. The performance is terrible compared to iteration, but it's not a big deal for what we're doing here. Then, use it to implement the `set` interface. Also, put a test file for this as well. You should be able to use pretty much all of the same tests from the hashset. The wonder of encapsulation.

Feel free to do the treeset before the hashset if you'd like. It'll certainly be much less code, and your hashset was written much earlier in your C career, so the code is probably pretty messy, and might still have bugs. Binary trees are usually pretty clean, and you might find it easier to just start over with something new.

Chapter 6

Miscellaneous C things

You actually know a pretty decent amount of C at this point. If you get everything so far, you're in a position where you could get by pretty well in a lot of projects. Now we're going to start talking about some cool stuff. I'm really not going to go into great detail on any of this or give exercises, this is just meant to be a jumping off point if you want to learn more about any of this.

6.1 Function Pointers

This is definitely the biggest issue in this section. The way code works (very broadly speaking for modern architechtures) is that the compiled binary code of your program is loaded into memory and control works its way down memory, executing instructions. There are specific sections of memory that correspond to the C functions that you write. That means your functions have a location in memory. See where this is going?

It turns out you can have a pointer to a function. This gives you a ton of power. One of the coolest features of functional programming languages is their treatment of functions as first class objects. You can have functions that take functions as arguments, return functions, functions stored in variables, whatever. With function pointers, you can have that same kind of power. The syntax is a little obtuse.

```c
// computes a float raised to an integer power
double power(double base, int exp)
{
  double res=1;
  for(int i=0; i<exp; i++) {
    res*=base;
  }
  return res;
}

int main(void)
{
  //declare a variable that holds the function ptr
  double (*func)(double, int);
  //get the pointer to the function
  func=&power;

  //call the function
  double x = (*func)(2.0,3);
  printf("%f\n", x); // prints 8.0

}
```

There's some pretty cool stuff you can do with this. Check out the standard library function `qsort`. You can quicksort an array of arbitrary elements according to whatever comparison metric you want, with a single function call. Pretty cool, right? Go into the `vecsort` directory. You'll find that I refactored the `vec3d` type from earlier, and in the `main` function, I sort an array of `vec3d` based on vector magnitude.

You can use function pointers all over the place. By having a function pointer field in a struct, you can now do things like have hashtables with custom hash functions and nuanced notions of equality.

6.2 Enumerated & Union Types

There's some other ways to define types in C. You're probably familiar with the concept of `enum` from other languages.

Say you really wanted a boolean type in C. You could do.

```c
enum bool {
  true,
  false
};
```

You'd probably want to do a typedef, like we do with struct, as well to get the nice syntax. What you might not be as familiar with is the concept of a union type. It looks a lot like a struct. Have a look at this code.

```
union things {
  int i;
  double y;
  char c;
};
```

Looks a lot like a struct, right? It does, but the difference is that as opposed to a struct, where all the declared data lives together, in a union, only one of the data types described lives in the union at a time. So it's size is simply the size of its largest element, so on my machine, sizeof(union things) would give 8, because of the double. A union alone is kind of hard to use. Have a look at the following. Assume the above union definition.

```
union things t;
t.i=5;
t.c='a';
t.y=2.0;
printf("%f\n", t.y); // prints 2.0
printf("%d\n", t.i); // prints ?
```

The value printed by the second print statement is undefined. What you'll realistically see is going to be some garbage value that comes from the bit representation of floating point numbers, which will be really weird. So if you want to actually use this in your code, you have to be super duper careful about which variant of the union is actually valid, because there's no way to tell which field is valid without having historical knowledge about the data.

Before we talk about how to organize this, I want to make an aside about how this stuff is used in other programming languages. The ML family of languages is a family of strongly-typed functional programming languages. In those languages, you can define types called "variant types" or "sum types." These are types that can have different forms for diffent types of data. It'd look like this

```
(*This is ML code NOT C*)
type things =
| Nothing
| Int of int
| Float of float
| Char of char
```

Where the capital letters are arbitrary "tags" that tell you what form the data has, and the things after the "of"s are the types of data contained when the variant has that form. Then when you work with a variable of that type, you can check which type it is using the tag and extract the data. This probably looks a lot like the above union, but there's two differences. First of all, there's actually a way to tell what kind of data the thing contains without having historical knowledge of that variable's "life story." So you get true, type-safe polymorphism, as opposed to what we've done with `void*`, where the only way to know that the `void*` you're casting points to what you want it to is careful programming. So how can we do this in C? You can combine enums and unions in a struct. Take a look

```
struct things {
  enum tags {
    Nothing,
    Int,
    Float,
    Char
  } tag;

  union data {
    int i;
    float f;
    char c;
  } dat;
}
```

This isn't quite as good as in ML, where the type checker guarantees that the data contained is the right type for the tag; you still have to make sure when you set the data that it has the right tag. But what this gets you is that when you get a variable of the `struct things` type, you can at least have some control flow based on the tag, and can then infer the type of the data comtained.

6.3 Exercises

At this point, it's really up to you whether you want to do this, but one idea is to refactor your hashtable so that its initialization function takes a function from `void*` to `unsigned` (or whatever type you used for your hash function in your original implementation) and use that hash function instead of the one you were using before.

Chapter 7

Appendix

7.1 Compiler Flags

This is going to be a quick rundown of the options you might see given to `gcc`. This is nowhere near exhaustive, it's really just a description of the options I use in the `Makefiles` for my examples, and some others that you might find useful. If you follow the structure of my makefiles, these are things you can put in your `COPTS`

- `-Wall` and `-Wextra` – By default, the compiler does not give you many warnings. Compilers generally give extremely useful warnings, and unless you really know what you're doing, there should be no warnings in your code. These options turn on various warnings. The former is absolutely essential, and you should always use it

- `-o <name>` – You use this to name your output executable. If this option is not specified, the executable will be named `a.out`. This looks like

  ```
  gcc myprog.c -o myprog.o
  ```

  If `myprog.c` is a valid C program, an executable `myprog.o` will be produced.

- `-std=<std>` – This simply sets the C standard you're compiling under. By default, this is `gnu90`, a standard specific to the GNU system based on `c90`. If you want to use any other standard, you need to specify it with this option.

- `-Werror` – This option causes warnings to be treated as errors and stop compilation. This is nice if you're not used taking compiller warnings seriously and need to get used to getting rid of them.

- `-g` and `-ggdb` – We'll talk more about this in the section on debugging, but this basically causes the compiler to attach a copy of your source code with the binary so that a debugger program can step through the source code instead of assembly. The former attaches the standard debugging information, while the latter does a little bit extra to help the GNU debugger do a little more. If you intend to use `gdb` or one of its frontends, you should probably use the latter.

- `-pedantic` and `-pedantic-errors` – every compiler must implement the full C standard for all the standards it supports, but most go beyond that. GCC, for example, has a great many language extensions, from a `typeof` operator to reason about the type of expressions, to a full set of thread-safe atomic functions. While these are nice, using them means that your code can only be compiled with `gcc`, which is not always desireable. For example, perhaps you are compiling with `gcc` to debug your program (since there is a rich set of tools to work with programs compiled with `gcc` on Linux), which is to be compiled for real with a different compiler, perhaps on a different platform. The only thing you can count on, in that case, is the C standard, so those options are very desireable. The former treats use of language extensions as a warning, the latter as an error.

- `-O, -O2` and `-O3` – Turn on various optimizations. It's an interesting fact, but a fact nonetheless, that human beings are really bad a writing efficient code. We write things in a way that make logical sense to us, not necessarily in the way that's most efficient. A lot of times, things can be rearranged, loops unrolled, function calls inlined, to get some more efficiency out of your code. Without any of these options, the compiler will not do these things, and will stick pretty closely to the code you write. This makes it ideal for debugging. By turning on successively higher levels of optimization, the compiler will attempt to use more and more optimization techniques, taking more time for compilation, but usually producing faster code. Be warned, optimized code looks really strange when you try to step through with the debugger.

Also, an interesting effect is that optimizations cause the compiler to "think harder" about your program, and will sometimes produce more warnings, so it's often a good idea to compile with optimizations until you know you want to use the debugger.

7.2 Make

If you've read the section about compiler options, and especially if you've read about using multiple files, you've probably realized that you have to type in a lot of stuff into the compiler. Nobody wants to sit there and type all of that stuff in over and over again every time they need to compile. So the natural thing to do is write a shell script or whatever to do the compilation. And that would work, but shell scripts are designed for general scripting, not compilation specifically.

GNU make is an ancient and venerable system for organizing compilation of large programs. It's almost like a programming language in and of itself. It can do a ton of really crazy things, and we'll be talking about approximately none of them. For the purposes of this text, I really could have made a build shell script for each example and had you run it. But `make` is used pretty ubiquitously in C projects, and it's good for you to get used to it.

Here's the brief rundown. You type `make` at the terminal, and the `make` program looks to see if there's a file named `Makefile` in the current working directory, and if there is, runs it. What does that mean? Makefiles have these things called rules. They look like names followed by a colon, with come commands indented on the following lines. So it looks like this.

```
rule_name:
        command
```

or, more realistically

```
myprog:
        gcc -Wall -O3 -o myprog.o myprog.c
```

Note that the indentation thing is not optional. Makefiles, unlike C, are highly sensitive to whitespace, and use things like indentations and newlines to determine which commands are which.

These rules are like functions. When they are called, the commands under them are run. This includes other rules. When you run `make rule` at the console, `make` looks for a rule named `rule`, and runs it if it exists. If you just type make, it will run the first rule it finds. So say you had the following as your Makefile.

```
prog1:
        gcc -Wall -O3 -o prog1.o prog1.c

prog2:
        gcc -Wall -O3 -o prog2.o prog2.c
```

You could type make prog1 or make prog2 to compile either of these programs. Typing make would be equivalent to the former. If you wanted to make it so that typing make would compile both at once, you could add the following

```
all: prog1 prog2

prog1:
        gcc -Wall -O3 -o prog1.o prog1.c

prog2:
        gcc -Wall -O3 -o prog2.o prog2.c
```

Now the rule all will run both of the other rules. Note that there is nothing special about the name all. It's just another rule, but since it's the first in the file, it will be run if you type make.

That's really all we're going to talk about in terms of actual functionality. But one other thing we should do is refactor this a little. It's a little repetitive having to type in all the compiler options and filenames and stuff. It also makes it a little annoying to change. It turns out that makefiles can have variables, which you can set and read in a way that should be apparent from the following example. This is a decent way of refactoring the above makefile. Somebody would probably think there's a better way, but this is how I'd do it.

```
CC=gcc
COPTS=-Wall -O3

EXEC1=prog1.o
SRC1=prog1.c

EXEC2=prog2.o
SRC2=prog2.c

all: prog1 prog2

prog1:
        $(CC) $(COPTS) -o $(EXEC1) $(SRC1)

prog2:
        $(CC) $(COPTS) -o $(EXEC1) $(SRC2)
```

The advantage of this is that the structure of the commands is all set up, and you only have to worry about content, and it's all right at the top of the file. So if you want to add another compiler flag, you can just write it in with the rest of the `COPTS` and not worry about where they go. If you have another source file in one of the projects, you can just write that in too. The makefiles for my example projects all follow this format, and now you should have the basic gist of what's going on.

7.3 Memory analysis (valgrind)

I have a feeling this will be pretty short. In fact, it probably doesn't deserve a whole section, but whatever. As I talk about in the section about memory allocation, working with allocated memory is hard. Your code can sometimes have bugs like memory leaks that are completely silent, but have serious consequences in the long run. In comes Valgrind. Valgrind is a program that analyzes the behavior of your program at runtime and looks for various memory problems like memory leaks, attempting to access freed blocks, accessing outside the bounds of allocated blocks, and that kind of thing. It's a really useful tool for working with allocated storage. Install Valgrind on Ubuntu by doing

```
sudo apt-get install valgrind
```

And if you have a program `prog.o` in your current working directory, use valgrind by doing

```
valgrind --leak-check=yes ./prog.o
```

You will notice that your program has a lot of weird-looking output on the console. Read it. If it says everything is good, congratulations. If not, the output will probably have some information about what's wrong that you can use to direct your debugging efforts.

7.4 Debugging

You will find, or perhaps already have found, that debugging C code is considerably harder than debugging code in high-level languages. There's a ton of reasons for this. In Java, when you overflow an array bound, the program exits gracefully, and gives you all kinds of nice information like a stack trace, the array that's messed up, and sometimes even the line in your source code that the problem occurred.

In C, there really aren't any well defined errors or exceptions. There's really just undefined behavior, which by definition means that you don't know what'll happen, so you have to figure everything out yourself. Obviously, you could use print statements to start to get a handle on things. But at a certain point, things get way too cluttered, and you're forgetting to print the right stuff, and you need to bring out the big guns.

A debugger is a special kind of program that takes your executable as an argument. It steps through your program, and allows you to stop and start and inspect your variables and the memory to try and get an idea of what's going on. Of course, in general, all you have is the machine code, which is not very interesting to look at. That's why you compile with debugging flags. If you do that, the compiler attaches a copy of the source code to the binary, so that the debugger can show you the actual source code you're looking at, and display the actual variables you'll recognize from your code, instead of meaningless register values.

The most important thing about using any debugger is controlling your place in the program. This means you need to tell it where to stop, when to go, and how far. Usually, "when to stop" is in the form of a breakpoint. The breakpoint is simply a place where, when the debugger is running, it is to stop and transfer control to the user. Once you're stopped, you can examine the program state in various ways. From there, you can tell the debugger to keep running the program until it hits another breakpoint, or you can step through it, perhaps stepping into function calls, perhaps stepping over them.

On Linux, the C debugger that's most common is the GNU debugger `gdb`. Like most of the GNU projects we've seen so far, it's a pretty ancient system, but it's stuck around because it works. Now, I'm not going to describe the details of how to use GDB. What I will say is that 'vanilla' gdb is a purely command-line entity. You interact with the debugger entirely via text commands and text output. While this works, and is good enough for many programmers far better than I, I personally believe that a graphical interface for a debugger is a big help. Many others aggree, so there are numerous front-ends that provide a graphical interface on top of GDB.

Emacs, for example, has a full GDB interface built into it, which you can access via `M-x gdb`. Look up the `gdb-many-windows` variable. If you're an Emacs user, this is great because you can debug and edit all in the same framework, and with some learning, you can get a really good workflow going. Another one I've had success with is `nemiver`, a project from Gnome, the GNU offshoot that created the most influential Linux desktop environment. What's great about it is that it's very portable. You can pretty much do a `sudo apt-get install nemiver`, do a `nemiver prog.o`, and you're up and running. Also, I feel it has a short learning curve, especially if you've used other graphical debuggers in the past. Choose whatever debugger you wish, but just pick one and learn it well.

It might seem like a lot of overhead to go through just to learn how to debug your program, but debugging is actually one of the most important skills you'll learn in learing C. Debuggers exist for every language imaginable, and are immensely useful in any large-scale application. Learning to use `gdb` will give you the baseline standard for what a debugger is. **Learning to use a debugger will make you a better programmer.**

Chapter 8

Next Steps

So now you've done this, what now? This text was an attempt to find a balance between being brief enough to get you up and running quickly, but still conveying a sense of the deeper concepts behind the language. I hope I accomplished part of that task. So just as there were two ideas behind this book, there are two ways you can go at this point.

One way you could go is to start coding in C. Figure out a project to code up and get cracking. A good place to start is data structures. They're pretty straightforward, at least in principle, and can be coded and tested without having to make too many complicated design decisions. And there's something extremely satisfying about writing up a really efficient data structure in a really efficient language. Other than that, it's up to you. If you want to play around with GUIs there's a lot to play around with, but be warned, there are very few substantial end-user applications written in C these days, and perhaps there's a reason for that.

The other way would be to continue with a more 'classroom-style' approach. *The C Programming Language,* lovingly known as *K&R* after its authors, is an amazing book and will teach you C top to bottom, and is still fairly short. It's also surprisingly readable. If you go through that cover to cover (full disclosure: I haven't), you'll have all the knowledge you need.

Either way, one thing that's good to learn about is doing multithreading in C. I didn't put it in this because language-level support for multithreading was only just added, and `gcc` only implemented it in version 4.9, which still hasn't made it to the Ubuntu repositories. Concurrency is an important aspect of modern programming; play around with it and see what you find.

Bibliography

[1] Do not trust random websites about c — ramblings. http://ramblings.implicit.net/posts/2014/5/3/do-not-trust-random-websites-about-c. Accessed: 2014-12-23.

[2] Randal Bryant and David R. O'Hallaron. Computer systems, a programmer's perspective. pages 623–625.

[3] ISO. The c11 standard. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[4] Olive Maudal. Deep c. http://www.slideshare.net/olvemaudal/deep-c.

[5] Dennis Ritchie and Brian Kernighan. *The C Progamming Language*. Prentice Hall, third edition, 1988.