Beej's Guide to C Programming

Brian “Beej Jorgensen” Hall

v0.9.14,

Contents

Chapter 1

Foreword

> *C is not a big language, and it is not well served by a big book.*
>
> --Brian W. Kernighan, Dennis M. Ritchie

No point in wasting words here, folks, let's jump straight into the C code:

```
E((ck?main((z?(stat(M,&t)?P+=a+'{'?0:3:
execv(M,k),a=G,i=P,y=G&255,
sprintf(Q,y/'@'-3?A(*L(V(%d+%d)+%d,0)
```

And they lived happily ever after. The End.

What's this? You say something's still not clear about this whole C programming language thing?

Well, to be quite honest, I'm not even sure what the above code does. It's a snippet from one of the entries in the 2001 International Obfuscated C Code Contest[1], a wonderful competition wherein the entrants attempt to write the most unreadable C code possible, with often surprising results.

The bad news is that if you're a beginner in this whole thing, all C code you see probably looks obfuscated! The good news is, it's not going to be that way for long.

What we'll try to do over the course of this guide is lead you from complete and utter sheer lost confusion on to the sort of enlightened bliss that can only be obtained through pure C programming. Right on.

In the old days, C was a simpler language. A good number of the features contained in this book and a *lot* of the features in the Library Reference volume didn't exist when K&R wrote the famous second edition of their book in 1988. Nevertheless, the language remains small at its core, and I hope I've presented it here in a way that starts with that simple core and builds outward.

And that's my excuse for writing such a hilariously large book for such a small, concise language.

1.1 Audience

This guide assumes that you've already got some programming knowledge under your belt from another language, such as Python[2], JavaScript[3], Java[4], Rust[5], Go[6], Swift[7], etc. (Objective-C[8] devs will have a par-

[1] https://www.ioccc.org/
[2] https://en.wikipedia.org/wiki/Python_(programming_language)
[3] https://en.wikipedia.org/wiki/JavaScript
[4] https://en.wikipedia.org/wiki/Java_(programming_language)
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)
[6] https://en.wikipedia.org/wiki/Go_(programming_language)
[7] https://en.wikipedia.org/wiki/Swift_(programming_language)
[8] https://en.wikipedia.org/wiki/Objective-C

ticularly easy time of it!)

We're going to assume you know what variables are, what loops do, how functions work, and so on.

If that's not you for whatever reason the best I can hope to provide is some honest entertainment for your reading pleasure. The only thing I can reasonably promise is that this guide won't end on a cliffhanger… or *will* it?

1.2 How to Read This Book

The guide is in two volumes, and this is the first: the tutorial volume!

The second volume is the library reference[9], and it's far more reference than tutorial.

If you're new, go through the tutorial part in order, generally. The higher you get in chapters, the less important it is to go in order.

And no matter your skill level, the reference part is there with complete examples of the standard library function calls to help refresh your memory whenever needed. Good for reading over a bowl of cereal or other time.

Finally, glancing at the index (if you're reading the print version), the reference section entries are italicized.

1.3 Platform and Compiler

I'll try to stick to Plain Ol'-Fashioned ISO-standard C[10]. Well, for the most part. Here and there I might go crazy and start talking about POSIX[11] or something, but we'll see.

Unix users (e.g. Linux, BSD, etc.) try running `cc` or `gcc` from the command line--you might already have a compiler installed. If you don't, search your distribution for installing `gcc` or `clang`.

Windows users should check out Visual Studio Community[12]. Or, if you're looking for a more Unix-like experience (recommended!), install WSL[13] and `gcc`.

Mac users will want to install XCode[14], and in particular the command line tools.

There are a lot of compilers out there, and virtually all of them will work for this book. And a C++ compiler will compile a lot of (but not all!) C code. Best use a proper C compiler if you can.

1.4 Official Homepage

This official location of this document is https://beej.us/guide/bgc/[15]. Maybe this'll change in the future, but it's more likely that all the other guides are migrated off Chico State computers.

1.5 Email Policy

I'm generally available to help out with email questions so feel free to write in, but I can't guarantee a response. I lead a pretty busy life and there are times when I just can't answer a question you have. When that's the case, I usually just delete the message. It's nothing personal; I just won't ever have the time to give the detailed answer you require.

[9] https://beej.us/guide/bgclr/
[10] https://en.wikipedia.org/wiki/ANSI_C
[11] https://en.wikipedia.org/wiki/POSIX
[12] https://visualstudio.microsoft.com/vs/community/
[13] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[14] https://developer.apple.com/xcode/
[15] https://beej.us/guide/bgc/

As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you're getting, and anything else you think might help me troubleshoot), you're much more likely to get a response.

If you don't get a response, hack on it some more, try to find the answer, and if it's still elusive, then write me again with the information you've found and hopefully it will be enough for me to help out.

Now that I've badgered you about how to write and not write me, I'd just like to let you know that I *fully* appreciate all the praise the guide has received over the years. It's a real morale boost, and it gladdens me to hear that it is being used for good! `:-)` Thank you!

1.6 Mirroring

You are more than welcome to mirror this site, whether publicly or privately. If you publicly mirror the site and want me to link to it from the main page, drop me a line at `beej@beej.us`.

1.7 Note for Translators

If you want to translate the guide into another language, write me at `beej@beej.us` and I'll link to your translation from the main page. Feel free to add your name and contact info to the translation.

Please note the license restrictions in the Copyright and Distribution section, below.

1.8 Copyright and Distribution

1.9 Dedication

The hardest things about writing these guides are:

- Learning the material in enough detail to be able to explain it
- Figuring out the best way to explain it clearly, a seemingly-endless iterative process
- Putting myself out there as a so-called *authority*, when really I'm just a regular human trying to make sense of it all, just like everyone else
- Keeping at it when so many other things draw my attention

A lot of people have helped me through this process, and I want to acknowledge those who have made this book possible.

- Everyone on the Internet who decided to help share their knowledge in one form or another. The free sharing of instructive information is what makes the Internet the great place that it is.
- The volunteers at cppreference.com[16] who provide the bridge that leads from the spec to the real world.
- The helpful and knowledgeable folks on comp.lang.c[17] and r/C_Programming[18] who got me through the tougher parts of the language.
- Everyone who submitted corrections and pull-requests on everything from misleading instructions to typos.

Thank you! ♥

[16] https://en.cppreference.com/
[17] https://groups.google.com/g/comp.lang.c
[18] https://www.reddit.com/r/C_Programming/

Chapter 2

Hello, World!

2.1 What to Expect from C

> ``*Where do these stairs go?*''
> ``*They go up.*''
>
> ---Ray Stantz and Peter Venkman, Ghostbusters

C is a low-level language.

It didn't use to be. Back in the day when people carved punch cards out of granite, C was an incredible way to be free of the drudgery of lower-level languages like assembly[1].

But now in these modern times, current-generation languages offer all kinds of features that didn't exist in 1972 when C was invented. This means C is a pretty basic language with not a lot of features. It can do *anything*, but it can make you work for it.

So why would we even use it today?

- As a learning tool: not only is C a venerable piece of computing history, but it is connected to the bare metal[2] in a way that present-day languages are not. When you learn C, you learn about how software interfaces with computer memory at a low level. There are no seatbelts. You'll write software that crashes, I assure you. And that's all part of the fun!
- As a useful tool: C still is used for certain applications, such as building operating systems[3] or in embedded systems[4]. (Though the Rust[5] programming language is eyeing both these fields!)

If you're familiar with another language, a lot of things about C are easy. C inspired many other languages, and you'll see bits of it in Go, Rust, Swift, Python, JavaScript, Java, and all kinds of other languages. Those parts will be familiar.

The one thing about C that hangs people up is *pointers*. Virtually everything else is familiar, but pointers are the weird one. The concept behind pointers is likely one you already know, but C forces you to be explicit about it, using operators you've likely never seen before.

It's especially insidious because once you *grok*[6] pointers, they're suddenly easy. But up until that moment, they're slippery eels.

[1] https://en.wikipedia.org/wiki/Assembly_language
[2] https://en.wikipedia.org/wiki/Bare_machine
[3] https://en.wikipedia.org/wiki/Operating_system
[4] https://en.wikipedia.org/wiki/Embedded_system
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)
[6] https://en.wikipedia.org/wiki/Grok

Everything else in C is just memorizing another way (or sometimes the same way!) of doing something you've done already. Pointers are the weird bit. And, arguably, even pointers are variations on a theme you're probably familiar with.

So get ready for a rollicking adventure as close to the core of the computer as you can get without assembly, in the most influential computer language of all time[7]. Hang on!

2.2 Hello, World!

This is the canonical example of a C program. Everyone uses it. (Note that the numbers to the left are for reader reference only, and are not part of the source code.)

```
1  /* Hello world program */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7      printf("Hello, World!\n");  // Actually do the work here
8  }
```

We're going to don our long-sleeved heavy-duty rubber gloves, grab a scalpel, and rip into this thing to see what makes it tick. So, scrub up, because here we go. Cutting very gently…

Let's get the easy thing out of the way: anything between the digraphs `/*` and `*/` is a comment and will be completely ignored by the compiler. Same goes for anything on a line after a `//`. This allows you to leave messages to yourself and others, so that when you come back and read your code in the distant future, you'll know what the heck it was you were trying to do. Believe me, you will forget; it happens.

Now, what is this `#include`? GROSS! Well, it tells the C Preprocessor to pull the contents of another file and insert it into the code right *there*.

Wait---what's a C Preprocessor? Good question. There are two stages[8] to compilation: the preprocessor and the compiler. Anything that starts with pound sign, or ``octothorpe'', (#) is something the preprocessor operates on before the compiler even gets started. Common *preprocessor directives*, as they're called, are `#include` and `#define`. More on that later.

Before we go on, why would I even begin to bother pointing out that a pound sign is called an octothorpe? The answer is simple: I think the word octothorpe is so excellently funny, I have to gratuitously spread its name around whenever I get the opportunity. Octothorpe. Octothorpe, octothorpe, octothorpe.

So *anyway*. After the C preprocessor has finished preprocessing everything, the results are ready for the compiler to take them and produce assembly code[9], machine code[10], or whatever it's about to do. Machine code is the ``language'' the CPU understands, and it can understand it *very rapidly*. This is one of the reasons C programs tend to be quick.

Don't worry about the technical details of compilation for now; just know that your source runs through the preprocessor, then the output of that runs through the compiler, then that produces an executable for you to run.

What about the rest of the line? What's `<stdio.h>`? That is what is known as a *header file*. It's the dot-h at the end that gives it away. In fact it's the ``Standard I/O'' (`stdio`) header file that you will grow to know and love. It gives us access to a bunch of I/O functionality[11]. For our demo program, we're outputting the string

[7] I know someone will fight me on that, but it's gotta be at least in the top three, right?
[8] Well, technically there are more than two, but hey, let's pretend there are two---ignorance is bliss, right?
[9] https://en.wikipedia.org/wiki/Assembly_language
[10] https://en.wikipedia.org/wiki/Machine_code
[11] Technically, it contains preprocessor directives and function prototypes (more on that later) for common input and output needs.

``Hello, World!", so we in particular need access to the `printf()` function to do this. The `<stdio.h>` file gives us this access. Basically, if we tried to use `printf()` without `#include <stdio.h>`, the compiler would have complained to us about it.

How did I know I needed to `#include <stdio.h>` for `printf()`? Answer: it's in the documentation. If you're on a Unix system, `man 3 printf` and it'll tell you right at the top of the man page what header files are required. Or see the reference section in this book. `:-)`

Holy moly. That was all to cover the first line! But, let's face it, it has been completely dissected. No mystery shall remain!

So take a breather…look back over the sample code. Only a couple easy lines to go.

Welcome back from your break! I know you didn't really take a break; I was just humoring you.

The next line is `main()`. This is the definition of the function `main()`; everything between the squirrelly braces (`{` and `}`) is part of the function definition.

(How do you *call* a different function, anyway? The answer lies in the `printf()` line, but we'll get to that in a minute.)

Now, the main function is a special one in many ways, but one way stands above the rest: it is the function that will be called automatically when your program starts executing. Nothing of yours gets called before `main()`. In the case of our example, this works fine since all we want to do is print a line and exit.

Oh, that's another thing: once the program executes past the end of `main()`, down there at the closing squirrelly brace, the program will exit, and you'll be back at your command prompt.

So now we know that that program has brought in a header file, `stdio.h`, and declared a `main()` function that will execute when the program is started. What are the goodies in `main()`?

I am so happy you asked. Really! We only have the one goodie: a call to the function `printf()`. You can tell this is a function call and not a function definition in a number of ways, but one indicator is the lack of squirrelly braces after it. And you end the function call with a semicolon so the compiler knows it's the end of the expression. You'll be putting semicolons after almost everything, as you'll see.

You're passing one argument to the function `printf()`: a string to be printed when you call it. Oh, yeah---we're calling a function! We rock! Wait, wait---don't get cocky. What's that crazy `\n` at the end of the string? Well, most characters in the string will print out just like they are stored. But there are certain characters that you can't print on screen well that are embedded as two-character backslash codes. One of the most popular is `\n` (read ``backslash-N" or simply ``newline") that corresponds to the *newline* character. This is the character that causes further printing to continue at the beginning of the next line instead of the current. It's like hitting return at the end of the line.

So copy that code into a file called `hello.c` and build it. On a Unix-like platform (e.g. Linux, BSD, Mac, or WSL), from the command line you'll build with a command like so:

```
gcc -o hello hello.c
```

(This means ``compile `hello.c`, and output an executable called `hello`".)

After that's done, you should have a file called `hello` that you can run with this command:

```
./hello
```

(The leading `./` tells the shell to ``run from the current directory".)

And see what happens:

```
Hello, World!
```

It's done and tested! Ship it!

2.3 Compilation Details

Let's talk a bit more about how to build C programs, and what happens behind the scenes there.

Like other languages, C has *source code*. But, depending on what language you're coming from, you might never have had to *compile* your source code into an *executable*.

Compilation is the process of taking your C source code and turning it into a program that your operating system can execute.

JavaScript and Python devs aren't used to a separate compilation step at all--though behind the scenes it's happening! Python compiles your source code into something called *bytecode* that the Python virtual machine can execute. Java devs are used to compilation, but that produces bytecode for the Java Virtual Machine.

When compiling C, *machine code* is generated. This is the 1s and 0s that can be executed directly and speedily by the CPU.

> Languages that typically aren't compiled are called *interpreted* languages. But as we mentioned with Java and Python, they also have a compilation step. And there's no rule saying that C can't be interpreted. (There are C interpreters out there!) In short, it's a bunch of gray areas. Compilation in general is just taking source code and turning it into another, more easily-executed form.

The C compiler is the program that does the compilation.

As we've already said, `gcc` is a compiler that's installed on a lot of Unix-like operating systems[12]. And it's commonly run from the command line in a terminal, but not always. You can run it from your IDE, as well.

So how do we do command line builds?

2.4 Building with `gcc`

If you have a source file called `hello.c` in the current directory, you can build that into a program called `hello` with this command typed in a terminal:

```
gcc -o hello hello.c
```

The `-o` means ``output to this file''[13]. And there's `hello.c` at the end, the name of the file we want to compile.

If your source is broken up into multiple files, you can compile them all together (almost as if they were one file, but the rules are actually more complex than that) by putting all the `.c` files on the command line:

```
gcc -o awesomegame ui.c characters.c npc.c items.c
```

and they'll all get built together into a big executable.

That's enough to get started---later we'll talk details about multiple source files, object files, and all kinds of fun stuff.

2.5 Building with `clang`

On Macs, the stock compiler isn't `gcc`---it's `clang`. But a wrapper is also installed so you can run `gcc` and have it still work.

You can also install the `gcc` compiler proper through Homebrew[14] or some other means.

[12] https://en.wikipedia.org/wiki/Unix

[13] If you don't give it an output filename, it will export to a file called `a.out` by default---this filename has its roots deep in Unix history.

[14] https://formulae.brew.sh/formula/gcc

2.6 Building from IDEs

If you're using an *Integrated Development Environment* (IDE), you probably don't have to build from the command line.

With Visual Studio, `CTRL-F7` will build, and `CTRL-F5` will run.

With VS Code, you can hit `F5` to run via the debugger. (You'll have to install the C/C++ Extension.)

With XCode, you can build with `COMMAND-B` and run with `COMMAND-R`. To get the command line tools, Google for ``XCode command line tools'' and you'll find instructions for installing them.

For getting started, I encourage you to also try to build from the command line---it's history!

2.7 C Versions

C has come a long way over the years, and it had many named version numbers to describe which dialect of the language you're using.

These generally refer to the year of the specification.

The most famous are C89, C99, C11, and C2x. We'll focus on the latter in this book.

But here's a more complete table:

Version	Description
K&R C	1978, the original. Named after Brian Kernighan and Dennis Ritchie. Ritchie designed and coded the language, and Kernighan co-authored the book on it. You rarely see original K&R code today. If you do, it'll look odd, like Middle English looks odd to modern English readers.
C89, ANSI C, C90	In 1989, the American National Standards Institute (ANSI) produced a C language specification that set the tone for C that persists to this day. A year later, the reins were handed to the International Organization for Standardization (ISO) that produced the identical C90.
C95	A rarely-mentioned addition to C89 that included wide character support.
C99	The first big overhaul with lots of language additions. The thing most people remember is the addition of `//`-style comments. This is the most popular version of C in use as of this writing.
C11	This major version update includes Unicode support and multi-threading. Be advised that if you start using these language features, you might be sacrificing portability with places that are stuck in C99 land. But, honestly, 1999 is getting to be a while back now.
C17, C18	Bugfix update to C11. C17 seems to be the official name, but the publication was delayed until 2018. As far as I can tell, these two are interchangeable, with C17 being preferred.
C2x	What's coming next! Expected to eventually become C23.

You can force GCC to use one of these standards with the `-std=` command line argument. If you want it to be picky about the standard, add `-pedantic`.

For example:

```
gcc -std=c11 -pedantic foo.c
```

For this book, I compile programs for C2x with all warnings set:

```
gcc -Wall -Wextra -std=c2x -pedantic foo.c
```

Chapter 3

Variables and Statements

> ``*It takes all kinds to make a world, does it not, Padre?''*
> ``*So it does, my son, so it does.''*
>
> ---Pirate Captain Thomas Bartholomew Red to the Padre, Pirates

There sure can be lotsa stuff in a C program.

Yup.

And for various reasons, it'll be easier for all of us if we classify some of the types of things you can find in a program, so we can be clear what we're talking about.

3.1 Variables

It's said that ``variables hold values". But another way to think about it is that a variable is a human-readable name that refers to some data in memory.

We're going to take a second here and take a peek down the rabbit hole that is pointers. Don't worry about it.

You can think of memory as a big array of bytes[1]. Data is stored in this ``array"[2]. If a number is larger than a single byte, it is stored in multiple bytes. Because memory is like an array, each byte of memory can be referred to by its index. This index into memory is also called an *address*, or a *location*, or a *pointer*.

When you have a variable in C, the value of that variable is in memory *somewhere*, at some address. Of course. After all, where else would it be? But it's a pain to refer to a value by its numeric address, so we make a name for it instead, and that's what the variable is.

The reason I'm bringing all this up is twofold:

1. It's going to make it easier to understand pointer variables later---they're variables that hold the address of other variables!
2. Also, it's going to make it easier to understand pointers later.

So a variable is a name for some data that's stored in memory at some address.

[1] A ``byte" is typically an 8-bit binary number. Think of it as an integer that can only hold the values from 0 to 255, inclusive. Technically, C allows bytes to be any number of bits and if you want to unambiguously refer to an 8-bit number, you should use the term *octet*. But programmers are going assume you mean 8-bits when you say ``byte" unless you specify otherwise.

[2] I'm seriously oversimplifying how modern memory works, here. But the mental model works, so please forgive me.

3.1.1 Variable Names

You can use any characters in the range 0-9, A-Z, a-z, and underscore for variable names, with the following rules:

- You can't start a variable with a digit 0-9.
- You can't start a variable name with two underscores.
- You can't start a variable name with an underscore followed by a capital A-Z.

For Unicode, just try it. There are some rules in the spec in §D.2 that talk about which Unicode codepoint ranges are allowed in which parts of identifiers, but that's too much to write about here and is probably something you'll never have to think about anyway.

3.1.2 Variable Types

Depending on which languages you already have in your toolkit, you might or might not be familiar with the idea of types. But C's kinda picky about them, so we should do a refresher.

Some example types, some of the most basic:

Type	Example	C Type
Integer	`3490`	`int`
Floating point	`3.14159`	`float`[3]
Character (single)	`'c'`	`char`
String	`"Hello, world!"`	`char *`[4]

C makes an effort to convert automatically between most numeric types when you ask it to. But other than that, all conversions are manual, notably between string and numeric.

Almost all of the types in C are variants on these types.

Before you can use a variable, you have to *declare* that variable and tell C what type the variable holds. Once declared, the type of variable cannot be changed later at runtime. What you set it to is what it is until it falls out of scope and is reabsorbed into the universe.

Let's take our previous ``Hello, world" code and add a couple variables to it:

```c
#include <stdio.h>

int main(void)
{
    int i;    // Holds signed integers, e.g. -3, -2, 0, 1, 10
    float f;  // Holds signed floating point numbers, e.g. -3.1416

    printf("Hello, World!\n");  // Ah, blessed familiarity
}
```

There! We've declared a couple of variables. We haven't used them yet, and they're both uninitialized. One holds an integer number, and the other holds a floating point number (a real number, basically, if you have a math background).

Uninitialized variables have indeterminate value[5]. They have to be initialized or else you must assume they contain some nonsense number.

[3] I'm lying here a little. Technically `3.14159` is of type `double`, but we're not there yet and I want you to associate `float` with ``Floating Point", and C will happily coerce that type into a `float`. In short, don't worry about it until later.

[4] Read this as ``pointer to a char" or ``char pointer". ``Char" for character. Though I can't find a study, it seems anecdotally most people pronounce this as ``char", a minority say ``car", and a handful say ``care". We'll talk more about pointers later.

[5] Colloquially, we say they have ``random" values, but they aren't truly---or even pseudo-truly---random numbers.

> This is one of the places C can ``get you". Much of the time, in my experience, the indeterminate value is zero… but it can vary from run to run! Never assume the value will be zero, even if you see it is. *Always* explicitly initialize variables to some value before you use them[6].

What's this? You want to store some numbers in those variables? Insanity!

Let's go ahead and do that:

```
int main(void)
{
    int i;

    i = 2; // Assign the value 2 into the variable i

    printf("Hello, World!\n");
}
```

Killer. We've stored a value. Let's print it.

We're going to do that by passing *two* amazing arguments to the `printf()` function. The first argument is a string that describes what to print and how to print it (called the *format string*), and the second is the value to print, namely whatever is in the variable `i`.

`printf()` hunts through the format string for a variety of special sequences which start with a percent sign (`%`) that tell it what to print. For example, if it finds a `%d`, it looks to the next parameter that was passed, and prints it out as an integer. If it finds a `%f`, it prints the value out as a float. If it finds a `%s`, it prints a string.

As such, we can print out the value of various types like so:

```
#include <stdio.h>

int main(void)
{
    int i = 2;
    float f = 3.14;
    char *s = "Hello, world!";  // char * ("char pointer") is the string type

    printf("%s  i = %d and f = %f!\n", s, i, f);
}
```

And the output will be:

```
Hello, world!  i = 2 and f = 3.140000!
```

In this way, `printf()` might be similar to various types of format strings or parameterized strings in other languages you're familiar with.

3.1.3 Boolean Types

C has Boolean types, true or false?

`1`!

Historically, C didn't have a Boolean type, and some might argue it still doesn't.

In C, `0` means ``false", and non-zero means ``true".

So `1` is true. And `-37` is true. And `0` is false.

[6] This isn't strictly 100% true. When we get to learning about static storage duration, you'll find the some variables are initialized to zero automatically. But the safe thing to do is always initialize them.

You can just declare Boolean types as `int`s:

```
int x = 1;

if (x) {
    printf("x is true!\n");
}
```

If you `#include <stdbool.h>`, you also get access to some symbolic names that might make things look more familiar, namely a `bool` type and `true` and `false` values:

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

int main(void) {
    bool x = true;

    if (x) {
        printf("x is true!\n");
    }
}
```

But these are identical to using integer values for true and false. They're just a facade to make things look nice.

3.2 Operators and Expressions

C operators should be familiar to you from other languages. Let's blast through some of them here.

(There are a bunch more details than this, but we're going to do enough in this section to get started.)

3.2.1 Arithmetic

Hopefully these are familiar:

```
i = i + 3;  // Addition (+) and assignment (=) operators, add 3 to i
i = i - 8;  // Subtraction, subtract 8 from i
i = i * 9;  // Multiplication
i = i / 2;  // Division
i = i % 5;  // Modulo (division remainder)
```

There are shorthand variants for all of the above. Each of those lines could more tersely be written as:

```
i += 3;  // Same as "i = i + 3", add 3 to i
i -= 8;  // Same as "i = i - 8"
i *= 9;  // Same as "i = i * 9"
i /= 2;  // Same as "i = i / 2"
i %= 5;  // Same as "i = i % 5"
```

There is no exponentiation. You'll have to use one of the `pow()` function variants from `math.h`.

Let's get into some of the weirder stuff you might not have in your other languages!

3.2.2 Ternary Operator

C also includes the *ternary operator*. This is an expression whose value depends on the result of a conditional embedded in it.

```
// If x > 10, add 17 to y. Otherwise add 37 to y.

y += x > 10? 17: 37;
```

What a mess! You'll get used to it the more you read it. To help out a bit, I'll rewrite the above expression using `if` statements:

```
// This expression:

y += x > 10? 17: 37;

// is equivalent to this non-expression:

if (x > 10)
    y += 17;
else
    y += 37;
```

Compare those two until you see each of the components of the ternary operator.

Or, another example that prints if a number stored in `x` is odd or even:

```
printf("The number %d is %s.\n", x, x % 2 == 0? "even": "odd");
```

The `%s` format specifier in `printf()` means print a string. If the expression `x % 2` evaluates to `0`, the value of the entire ternary expression evaluates to the string `"even"`. Otherwise it evaluates to the string `"odd"`. Pretty cool!

It's important to note that the ternary operator isn't flow control like the `if` statement is. It's just an expression that evaluates to a value.

3.2.3 Pre-and-Post Increment-and-Decrement

Now, let's mess with another thing that you might not have seen.

These are the legendary post-increment and post-decrement operators:

```
i++;        // Add one to i (post-increment)
i--;        // Subtract one from i (post-decrement)
```

Very commonly, these are just used as shorter versions of:

```
i += 1;        // Add one to i
i -= 1;        // Subtract one from i
```

but they're more subtly different than that, the clever scoundrels.

Let's take a look at this variant, pre-increment and pre-decrement:

```
++i;        // Add one to i (pre-increment)
--i;        // Subtract one from i (pre-decrement)
```

With pre-increment and pre-decrement, the value of the variable is incremented or decremented *before* the expression is evaluated. Then the expression is evaluated with the new value.

With post-increment and post-decrement, the value of the expression is first computed with the value as-is, and *then* the value is incremented or decremented after the value of the expression has been determined.

You can actually embed them in expressions, like this:

```
i = 10;
j = 5 + i++;  // Compute 5 + i, _then_ increment i

printf("%d, %d\n", i, j);  // Prints 11, 15
```

Let's compare this to the pre-increment operator:

```
i = 10;
j = 5 + ++i;  // Increment i, _then_ compute 5 + i

printf("%d, %d\n", i, j);  // Prints 11, 16
```

This technique is used frequently with array and pointer access and manipulation. It gives you a way to use the value in a variable, and also increment or decrement that value before or after it is used.

But by far the most common place you'll see this is in a `for` loop:

```
for (i = 0; i < 10; i++)
    printf("i is %d\n", i);
```

But more on that later.

3.2.4 The Comma Operator

This is an uncommonly-used way to separate expressions that will run left to right:

```
x = 10, y = 20;  // First assign 10 to x, then 20 to y
```

Seems a bit silly, since you could just replace the comma with a semicolon, right?

```
x = 10; y = 20;  // First assign 10 to x, then 20 to y
```

But that's a little different. The latter is two separate expressions, while the former is a single expression!

With the comma operator, the value of the comma expression is the value of the rightmost expression:

```
x = (1, 2, 3);

printf("x is %d\n", x);  // Prints 3, because 3 is rightmost in the comma list
```

But even that's pretty contrived. One common place the comma operator is used is in `for` loops to do multiple things in each section of the statement:

```
for (i = 0, j = 10; i < 100; i++, j++)
    printf("%d, %d\n", i, j);
```

We'll revisit that later.

3.2.5 Conditional Operators

For Boolean values, we have a raft of standard operators:

```
a == b;  // True if a is equivalent to b
a != b;  // True if a is not equivalent to b
a < b;   // True if a is less than b
a > b;   // True if a is greater than b
a <= b;  // True if a is less than or equal to b
a >= b;  // True if a is greater than or equal to b
```

Don't mix up assignment = with comparison ==! Use two equals to compare, one to assign.

We can use the comparison expressions with `if` statements:

```
if (a <= 10)
    printf("Success!\n");
```

3.2.6 Boolean Operators

We can chain together or alter conditional expressions with Boolean operators for *and*, *or*, and *not*.

Operator	Boolean meaning
&&	and
||	or
!	not

An example of Boolean ``and'':

```
// Do something if x less than 10 and y greater than 20:

if (x < 10 && y > 20)
    printf("Doing something!\n");
```

An example of Boolean ``not'':

```
if (!(x < 12))
    printf("x is not less than 12\n");
```

`!` has higher precedence than the other Boolean operators, so we have to use parentheses in that case.

Of course, that's just the same as:

```
if (x >= 12)
    printf("x is not less than 12\n");
```

but I needed the example!

3.2.7 The `sizeof` Operator

This operator tells you the size (in bytes) that a particular variable or data type uses in memory.

More particularly, it tells you the size (in bytes) that the *type of a particular expression* (which might be just a single variable) uses in memory.

This can be different on different systems, except for `char` and its variants (which are always 1 byte).

And this might not seem very useful now, but we'll be making references to it here and there, so it's worth covering.

Since this computes the number of bytes needed to store a type, you might think it would return an `int`. Or… since the size can't be negative, maybe an `unsigned`?

But it turns out C has a special type to represent the return value from `sizeof`. It's `size_t`, pronounced ``*size tee*''[7]. All we know is that it's an unsigned integer type that can hold the size in bytes of anything you can give to `sizeof`.

`size_t` shows up a lot of different places where counts of things are passed or returned. Think of it as a value that represents a count.

You can take the `sizeof` a variable or expression:

[7] The `_t` is short for `type`.

```
int a = 999;

// %zu is the format specifier for type size_t

printf("%zu\n", sizeof a);      // Prints 4 on my system
printf("%zu\n", sizeof(2 + 7)); // Prints 4 on my system
printf("%zu\n", sizeof 3.14);   // Prints 8 on my system

// If you need to print out negative size_t values, use %zd
```

Remember: it's the size in bytes of the *type* of the expression, not the size of the expression itself. That's why the size of `2+7` is the same as the size of `a`---they're both type `int`. We'll revisit this number `4` in the very next block of code…

…Where we'll see you can take the `sizeof` a type (note the parentheses are required around a type name, unlike an expression):

```
printf("%zu\n", sizeof(int));   // Prints 4 on my system
printf("%zu\n", sizeof(char));  // Prints 1 on all systems
```

It's important to note that `sizeof` is a *compile-time* operation[8]. The result of the expression is determined entirely at compile-time, not at runtime.

We'll make use of this later on.

3.3 Flow Control

Booleans are all good, but of course we're nowhere if we can't control program flow. Let's take a look at a number of constructs: `if`, `for`, `while`, and `do-while`.

First, a general forward-looking note about statements and blocks of statements brought to you by your local friendly C developer:

After something like an `if` or `while` statement, you can either put a single statement to be executed, or a block of statements to all be executed in sequence.

Let's start with a single statement:

```
if (x == 10) printf("x is 10\n");
```

This is also sometimes written on a separate line. (Whitespace is largely irrelevant in C---it's not like Python.)

```
if (x == 10)
    printf("x is 10\n");
```

But what if you want multiple things to happen due to the conditional? You can use squirrelly braces to mark a *block* or *compound statement*.

```
if (x == 10) {
    printf("x is 10\n");
    printf("And also this happens when x is 10\n");
}
```

It's a really common style to *always* use squirrelly braces even if they aren't necessary:

```
if (x == 10) {
    printf("x is 10\n");
}
```

[8] Except for with variable length arrays---but that's a story for another time.

Some devs feel the code is easier to read and avoids errors like this where things visually look like they're in the `if` block, but actually they aren't.

```
// BAD ERROR EXAMPLE

if (x == 10)
    printf("This happens if x is 10\n");
    printf("This happens ALWAYS\n");  // Surprise!! Unconditional!
```

`while` and `for` and the other looping constructs work the same way as the examples above. If you want to do multiple things in a loop or after an `if`, wrap them up in squirrelly braces.

In other words, the `if` is going to run the one thing after the `if`. And that one thing can be a single statement or a block of statements.

3.3.1 The `if-else` statement

We've already been using `if` for multiple examples, since it's likely you've seen it in a language before, but here's another:

```
int i = 10;

if (i > 10) {
    printf("Yes, i is greater than 10.\n");
    printf("And this will also print if i is greater than 10.\n");
}

if (i <= 10) printf("i is less than or equal to 10.\n");
```

In the example code, the message will print if `i` is greater than 10, otherwise execution continues to the next line. Notice the squirrley braces after the `if` statement; if the condition is true, either the first statement or expression right after the if will be executed, or else the collection of code in the squirlley braces after the `if` will be executed. This sort of *code block* behavior is common to all statements.

Of course, because C is fun this way, you can also do something if the condition is false with an `else` clause on your `if`:

```
int i = 99;

if (i == 10)
    printf("i is 10!\n");
else {
    printf("i is decidedly not 10.\n");
    printf("Which irritates me a little, frankly.\n");
}
```

And you can even cascade these to test a variety of conditions, like this:

```
int i = 99;

if (i == 10)
    printf("i is 10!\n");

else if (i == 20)
    printf("i is 20!\n");

else if (i == 99) {
    printf("i is 99! My favorite\n");
```

```
    printf("I can't tell you how happy I am.\n");
    printf("Really.\n");
}

else
    printf("i is some crazy number I've never heard of.\n");
```

Though if you're going that route, be sure to check out the `switch` statement for a potentially better solution. The catch is `switch` only works with equality comparisons with constant numbers. The above `if-else` cascade could check inequality, ranges, variables, or anything else you can craft in a conditional expression.

3.3.2 The `while` statement

`while` is your average run-of-the-mill looping construct. Do a thing while a condition expression is true.

Let's do one!

```
// Print the following output:
//
//   i is now 0!
//   i is now 1!
//   [ more of the same between 2 and 7 ]
//   i is now 8!
//   i is now 9!

int i = 0;

while (i < 10) {
    printf("i is now %d!\n", i);
    i++;
}

printf("All done!\n");
```

That gets you a basic loop. C also has a `for` loop which would have been cleaner for that example.

A not-uncommon use of `while` is for infinite loops where you repeat while true:

```
while (1) {
    printf("1 is always true, so this repeats forever.\n");
}
```

3.3.3 The `do-while` statement

So now that we've gotten the `while` statement under control, let's take a look at its closely related cousin, `do-while`.

They are basically the same, except if the loop condition is false on the first pass, `do-while` will execute once, but `while` won't execute at all. In other words, the test to see whether or not to execute the block happens at the *end* of the block with `do-while`. It happens at the *beginning* of the block with `while`.

Let's see by example:

```
// Using a while statement:

i = 10;
```

```
// this is not executed because i is not less than 10:
while(i < 10) {
    printf("while: i is %d\n", i);
    i++;
}

// Using a do-while statement:

i = 10;

// this is executed once, because the loop condition is not checked until
// after the body of the loop runs:

do {
    printf("do-while: i is %d\n", i);
    i++;
} while (i < 10);

printf("All done!\n");
```

Notice that in both cases, the loop condition is false right away. So in the `while`, the loop fails, and the following block of code is never executed. With the `do-while`, however, the condition is checked *after* the block of code executes, so it always executes at least once. In this case, it prints the message, increments `i`, then fails the condition, and continues to the ``All done!'' output.

The moral of the story is this: if you want the loop to execute at least once, no matter what the loop condition, use `do-while`.

All these examples might have been better done with a `for` loop. Let's do something less deterministic---repeat until a certain random number comes up!

```
#include <stdio.h>   // For printf
#include <stdlib.h>  // For rand

int main(void)
{
    int r;

    do {
        r = rand() % 100; // Get a random number between 0 and 99
        printf("%d\n", r);
    } while (r != 37);    // Repeat until 37 comes up
}
```

Side note: did you run that more than once? If you did, did you notice the same sequence of numbers came up again. And again. And again? This is because `rand()` is a pseudorandom number generator that must be *seeded* with a different number in order to generate a different sequence. Look up the `srand()`[9] function for more details.

3.3.4 The `for` statement

Welcome to one of the most popular loops in the world! The `for` loop!

This is a great loop if you know the number of times you want to loop in advance.

[9] https://beej.us/guide/bgclr/html/split/stdlib.html#man-srand

You could do the same thing using just a `while` loop, but the `for` loop can help keep the code cleaner.

Here are two pieces of equivalent code---note how the `for` loop is just a more compact representation:

```
// Print numbers between 0 and 9, inclusive...

// Using a while statement:

i = 0;
while (i < 10) {
    printf("i is %d\n", i);
    i++;
}

// Do the exact same thing with a for-loop:

for (i = 0; i < 10; i++) {
    printf("i is %d\n", i);
}
```

That's right, folks---they do exactly the same thing. But you can see how the `for` statement is a little more compact and easy on the eyes. (JavaScript users will fully appreciate its C origins at this point.)

It's split into three parts, separated by semicolons. The first is the initialization, the second is the loop condition, and the third is what should happen at the end of the block if the loop condition is true. All three of these parts are optional.

```
for (initialize things; loop if this is true; do this after each loop)
```

Note that the loop will not execute even a single time if the loop condition starts off false.

> **`for`-loop fun fact!**
>
> You can use the comma operator to do multiple things in each clause of the `for` loop!
>
> ```
> for (i = 0, j = 999; i < 10; i++, j--) {
>     printf("%d, %d\n", i, j);
> }
> ```

An empty `for` will run forever:

```
for(;;) {  // "forever"
    printf("I will print this again and again and again\n" );
    printf("for all eternity until the heat-death of the universe.\n");

    printf("Or until you hit CTRL-C.\n");
}
```

3.3.5 The `switch` Statement

Depending on what languages you're coming from, you might or might not be familiar with `switch`, or C's version might even be more restrictive than you're used to. This is a statement that allows you to take a variety of actions depending on the value of an integer expression.

Basically, it evaluates an expression to an integer value, jumps to the `case` that corresponds to that value. Execution resumes from that point. If a `break` statement is encountered, then execution jumps out of the `switch`.

Let's do an example where the user enters a number of goats and we print out a gut-feel of how many goats that is.

```c
#include <stdio.h>

int main(void)
{
    int goat_count;

    printf("Enter a goat count: ");
    scanf("%d", &goat_count);       // Read an integer from the keyboard

    switch (goat_count) {
        case 0:
            printf("You have no goats.\n");
            break;

        case 1:
            printf("You have a singular goat.\n");
            break;

        case 2:
            printf("You have a brace of goats.\n");
            break;

        default:
            printf("You have a bona fide plethora of goats!\n");
            break;
    }
}
```

In that example, if the user enters, say, `2`, the `switch` will jump to the `case 2` and execute from there. When (if) it hits a `break`, it jumps out of the `switch`.

Also, you might see that `default` label there at the bottom. This is what happens when no cases match.

Every `case`, including `default`, is optional. And they can occur in any order, but it's really typical for `default`, if any, to be listed last.

So the whole thing acts like an `if-else` cascade:

```c
if (goat_count == 0)
    printf("You have no goats.\n");
else if (goat_count == 1)
    printf("You have a singular goat.\n");
else if (goat_count == 2)
    printf("You have a brace of goats.\n");
else
    printf("You have a bona fide plethora of goats!\n");
```

With some key differences:

- `switch` is often faster to jump to the correct code (though the spec makes no such guarantee).
- `if-else` can do things like relational conditionals like `<` and `>=` and floating point and other types, while `switch` cannot.

There's one more neat thing about switch that you sometimes see that is quite interesting: *fall through*.

Remember how `break` causes us to jump out of the switch?

Well, what happens if we *don't* `break`?

Turns out we just keep on going into the next `case`! Demo!

```
switch (x) {
    case 1:
        printf("1\n");
        // Fall through!
    case 2:
        printf("2\n");
        break;
    case 3:
        printf("3\n");
        break;
}
```

If `x == 1`, this `switch` will first hit `case 1`, it'll print the `1`, but then it just continues on to the next line of code… which prints `2`!

And then, at last, we hit a `break` so we jump out of the `switch`.

if `x == 2`, then we just hit the `case 2`, print `2`, and `break` as normal.

Not having a `break` is called *fall through*.

ProTip: *ALWAYS* put a comment in the code where you intend to fall through, like I did above. It will save other programmers from wondering if you meant to do that.

In fact, this is one of the common places to introduce bugs in C programs: forgetting to put a `break` in your `case`. You gotta do it if you don't want to just roll into the next case[10].

Earlier I said that `switch` works with integer types---keep it that way. Don't use floating point or string types in there. One loophole-ish thing here is that you can use character types because those are secretly integers themselves. So this is perfectly acceptable:

```
char c = 'b';

switch (c) {
    case 'a':
        printf("It's 'a'!\n");
        break;

    case 'b':
        printf("It's 'b'!\n");
        break;

    case 'c':
        printf("It's 'c'!\n");
        break;
}
```

Finally, you can use `enum`s in `switch` since they are also integer types. But more on that in the `enum` chapter.

[10]This was considered such a hazard that the designers of the Go Programming Language made `break` the default; you have to explicitly use Go's `fallthrough` statement if you want to fall into the next case.

Chapter 4

Functions

> ``Sir, not in an environment such as this. That's why I've also been programmed for over thirty secondary functions that---''
>
> ---C3PO, before being rudely interrupted, reporting a now-unimpressive number of additional functions, *Star Wars* script

Very much like other languages you're used to, C has the concept of *functions*.

Functions can accept a variety of *arguments* and return a value. One important thing, though: the arguments and return value types are predeclared---because that's how C likes it!

Let's take a look at a function. This is a function that takes an `int` as an argument, and returns an `int`.

```
#include <stdio.h>

int plus_one(int n)  // The "definition"
{
    return n + 1;
}

```

The `int` before the `plus_one` indicates the return type.

The `int n` indicates that this function takes one `int` argument, stored in *parameter* `n`. A parameter is a special type of local variable into which the arguments are copied.

I'm going to drive home the point that the arguments are copied into the parameters, here. Lots of things in C are easier to understand if you know that the parameter is a *copy* of the argument, not the argument itself. More on that in a minute.

Continuing the program down into `main()`, we can see the call to the function, where we assign the return value into local variable `j`:

```
int main(void)
{
    int i = 10, j;

    j = plus_one(i);  // The "call"

    printf("i + 1 is %d\n", j);
}
```

> Before I forget, notice that I defined the function before I used it. If I hadn't done that, the compiler wouldn't know about it yet when it compiles `main()` and it would have given an unknown function call error. There is a more proper way to do the above code with *function prototypes*, but we'll talk about that later.

Also notice that `main()` is a function!

It returns an `int`.

But what's this `void` thing? This is a keyword that's used to indicate that the function accepts no arguments.

You can also return `void` to indicate that you don't return a value:

```
#include <stdio.h>

// This function takes no arguments and returns no value:

void hello(void)
{
    printf("Hello, world!\n");
}

int main(void)
{
    hello();  // Prints "Hello, world!"
}
```

4.1 Passing by Value

I'd mentioned earlier that when you pass an argument to a function, a copy of that argument gets made and stored in the corresponding parameter.

If the argument is a variable, a copy of the value of that variable gets made and stored in the parameter.

More generally, the entire argument expression is evaluated and its value determined. That value is copied to the parameter.

In any case, the value in the parameter is its own thing. It is independent of whatever values or variables you used as arguments when you made the function call.

So let's look at an example here. Study it and see if you can determine the output before running it:

```
#include <stdio.h>

void increment(int a)
{
    a++;
}

int main(void)
{
    int i = 10;

    increment(i);

    printf("i == %d\n", i);  // What does this print?
}
```

At first glance, it looks like `i` is `10`, and we pass it to the function `increment()`. There the value gets incremented, so when we print it, it must be `11`, right?

> ``Get used to disappointment.''
>
> ---Dread Pirate Roberts, *The Princess Bride*

But it's not `11`---it prints `10`! How?

It's all about the fact that the expressions you pass to functions get *copied* onto their corresponding parameters. The parameter is a copy, not the original.

So `i` is `10` out in `main()`. And we pass it to `increment()`. The corresponding parameter is called `a` in that function.

And the copy happens, as if by assignment. Loosely, `a = i`. So at that point, `a` is `10`. And out in `main()`, `i` is also `10`.

Then we increment `a` to `11`. But we're not touching `i` at all! It remains `10`.

Finally, the function is complete. All its local variables are discarded (bye, `a`!) and we return to `main()`, where `i` is still `10`.

And we print it, getting `10`, and we're done.

This is why in the previous example with the `plus_one()` function, we `return`ed the locally modified value so that we could see it again in `main()`.

Seems a little bit restrictive, huh? Like you can only get one piece of data back from a function, is what you're thinking. There is, however, another way to get data back; C folks call it *passing by reference* and that's a story we'll tell another time.

But no fancy-schmancy name will distract you from the fact that *EVERYTHING* you pass to a function *WITHOUT EXCEPTION* is copied into its corresponding parameter, and the function operates on that local copy, *NO MATTER WHAT*. Remember that, even when we're talking about this so-called passing by reference.

4.2 Function Prototypes

So if you recall back in the ice age a few sections ago, I mentioned that you had to define the function before you used it, otherwise the compiler wouldn't know about it ahead of time, and would bomb out with an error.

This isn't quite strictly true. You can notify the compiler in advance that you'll be using a function of a certain type that has a certain parameter list. That way the function can be defined anywhere (even in a different file), as long as the *function prototype* has been declared before you call that function.

Fortunately, the function prototype is really quite easy. It's merely a copy of the first line of the function definition with a semicolon tacked on the end for good measure. For example, this code calls a function that is defined later, because a prototype has been declared first:

```
#include <stdio.h>

int foo(void);  // This is the prototype!

int main(void)
{
    int i;

    // We can call foo() here before it's definition because the
    // prototype has already been declared, above!

```

```
    i = foo();

    printf("%d\n", i);  // 3490
}

int foo(void)  // This is the definition, just like the prototype!
{
    return 3490;
}
```

If you don't declare your function before you use it (either with a prototype or its definition), you're performing something called an *implicit declaration*. This was allowed in the first C standard (C89), and that standard has rules about it, but is no longer allowed today. And there is no legitimate reason to rely on it in new code.

You might notice something about the sample code we've been using… That is, we've been using the good old `printf()` function without defining it or declaring a prototype! How do we get away with this lawlessness? We don't, actually. There is a prototype; it's in that header file `stdio.h` that we included with `#include`, remember? So we're still legit, officer!

4.3 Empty Parameter Lists

You might see these from time to time in older code, but you shouldn't ever code one up in new code. Always use `void` to indicate that a function takes no parameters. There's never[1] a reason to skip this in modern code.

If you're good at just remembering to put `void` in for empty parameter lists in functions and prototypes, you can skip the rest of this section.

There are two contexts for this:

- Omitting all parameters where the function is defined
- Omitting all parameters in a prototype

Let's look at a potential function definition first:

```
void foo()  // Should really have a `void` in there
{
    printf("Hello, world!\n");
}
```

While the spec spells out that the behavior in this instance is *as-if* you'd indicated `void` (C11 §6.7.6.3¶14), the `void` type is there for a reason. Use it.

But in the case of a function prototype, there is a *significant* difference between using `void` and not:

```
void foo();
void foo(void);  // Not the same!
```

Leaving `void` out of the prototype indicates to the compiler that there is no additional information about the parameters to the function. It effectively turns off all that type checking.

With a prototype **definitely** use `void` when you have an empty parameter list.

[1] Never say ``never".

Chapter 5

Pointers---Cower In Fear!

> ``*How do you get to Carnegie Hall?*''
> ``*Practice!*''
>
> ---20th-century joke of unknown origin

Pointers are one of the most feared things in the C language. In fact, they are the one thing that makes this language challenging at all. But why?

Because they, quite honestly, can cause electric shocks to come up through the keyboard and physically *weld* your arms permanently in place, cursing you to a life at the keyboard in this language from the 70s!

Really? Well, not really. I'm just trying to set you up for success.

Depending on what language you came from, you might already understand the concept of *references*, where a variable refers to an object of some type.

This is very much the same, except we have to be more explicit with C about when we're talking about the reference or the thing it refers to.

5.1 Memory and Variables

Computer memory holds data of all kinds, right? It'll hold `floats`, `ints`, or whatever you have. To make memory easy to cope with, each byte of memory is identified by an integer. These integers increase sequentially as you move up through memory[1]. You can think of it as a bunch of numbered boxes, where each box holds a byte[2] of data. Or like a big array where each element holds a byte, if you come from a language with arrays. The number that represents each box is called its *address*.

Now, not all data types use just a byte. For instance, an `int` is often four bytes, as is a `float`, but it really depends on the system. You can use the `sizeof` operator to determine how many bytes of memory a certain type uses.

```
// %zu is the format specifier for type size_t

printf("an int uses %zu bytes of memory\n", sizeof(int));

// That prints "4" for me, but can vary by system.
```

[1] Typically. I'm sure there are exceptions out there in the dark corridors of computing history.

[2] A byte is a number made up of no more than 8 binary digits, or *bits* for short. This means in decimal digits just like grandma used to use, it can hold an unsigned number between 0 and 255, inclusive.

> **Memory Fun Facts**: When you have a data type (like your typical `int`) that uses more than a byte of memory, the bytes that make up the data are always adjacent to one another in memory. Sometimes they're in the order that you expect, and sometimes they're not[3]. While C doesn't guarantee any particular memory order (it's platform-dependent), it's still generally possible to write code in a way that's platform-independent where you don't have to even consider these pesky byte orderings.

So *anyway*, if we can get on with it and get a drum roll and some foreboding music playing for the definition of a pointer, *a pointer is a variable that holds an address*. Imagine the classical score from 2001: A Space Odyssey at this point. Ba bum ba bum ba bum BAAAAH!

Ok, so maybe a bit overwrought here, yes? There's not a lot of mystery about pointers. They are the address of data. Just like an `int` variable can hold the value `12`, a pointer variable can hold the address of data.

This means that all these things mean the same thing, i.e. a number that represents a point in memory:

- Index into memory (if you're thinking of memory like a big array)
- Address
- Location

I'm going to use these interchangeably. And yes, I just threw *location* in there because you can never have enough words that mean the same thing.

And a pointer variable holds that address number. Just like a `float` variable might hold `3.14159`.

Imagine you have a bunch of Post-it® notes all numbered in sequence with their address. (The first one is at index numbered `0`, the next at index `1`, and so on.)

In addition to the number representing their positions, you can also write another number of your choice on each. It could be the number of dogs you have. Or the number of moons around Mars…

…Or, *it could be the index of another Post-it note!*

If you have written the number of dogs you have, that's just a regular variable. But if you wrote the index of another Post-it in there, *that's a pointer*. It points to the other note!

Another analogy might be with house addresses. You can have a house with certain qualities, yard, metal roof, solar, etc. Or you could have the address of that house. The address isn't the same as the house itself. One's a full-blown house, and the other is just a few lines of text. But the address of the house is a *pointer* to that house. It's not the house itself, but it tells you where to find it.

And we can do the same thing in the computer with data. You can have a data variable that's holding some value. And that value is in memory at some address. And you could have a different *pointer variable* hold the address of that data variable.

It's not the data variable itself, but, like with a house address, it tells us where to find it.

When we have that, we say we have a ``pointer to" that data. And we can follow the pointer to access the data itself.

(Though it doesn't seem particularly useful yet, this all becomes indispensable when used with function calls. Bear with me until we get there.)

So if we have an `int`, say, and we want a pointer to it, what we want is some way to get the address of that `int`, right? After all, the pointer just holds the *address of* the data. What operator do you suppose we'd use to find the *address of* the `int`?

Well, by a shocking surprise that must come as something of a shock to you, gentle reader, we use the `address-of` operator (which happens to be an ampersand: ``&")to find the address of the data. Ampersand.

[3]The order that bytes come in is referred to as the *endianness* of the number. The usual suspects are *big-endian* (with the most significant byte first) and *little-endian* (with the most-significant byte last), or, uncommonly now, *mixed-endian* (with the most-significant bytes somewhere else).

So for a quick example, we'll introduce a new *format specifier* for `printf()` so you can print a pointer. You know already how `%d` prints a decimal integer, yes? Well, `%p` prints a pointer. Now, this pointer is going to look like a garbage number (and it might be printed in hexadecimal[4] instead of decimal), but it is merely the index into memory the data is stored in. (Or the index into memory that the first byte of data is stored in, if the data is multi-byte.) In virtually all circumstances, including this one, the actual value of the number printed is unimportant to you, and I show it here only for demonstration of the `address-of` operator.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    printf("The value of i is %d\n", i);
    printf("And its address is %p\n", (void *)&i);

    // %p expects the argument to be a pointer to void
    // so we cast it to make the compiler happy.
}
```

On my computer, this prints:

```
The value of i is 10
And its address is 0x7ffddf7072a4
```

If you're curious, that hexadecimal number is 140,727,326,896,068 in decimal (base 10 just like Grandma used to use). That's the index into memory where the variable `i`'s data is stored. It's the address of `i`. It's the location of `i`. It's a pointer to `i`.

It's a pointer because it lets you know where `i` is in memory. Like a home address written on a scrap of paper tells you where you can find a particular house, this number indicates to us where in memory we can find the value of `i`. It points to `i`.

Again, we don't really care what the address's exact number is, generally. We just care that it's a pointer to `i`.

5.2 Pointer Types

So… this is all well and good. You can now successfully take the address of a variable and print it on the screen. There's a little something for the ol' resume, right? Here's where you grab me by the scruff of the neck and ask politely what the frick pointers are good for.

Excellent question, and we'll get to that right after these messages from our sponsor.

> ACME ROBOTIC HOUSING UNIT CLEANING SERVICES. YOUR HOMESTEAD WILL BE DRAMATICALLY IMPROVED OR YOU WILL BE TERMINATED. MESSAGE ENDS.

Welcome back to another installment of Beej's Guide. When we met last we were talking about how to make use of pointers. Well, what we're going to do is store a pointer off in a variable so that we can use it later. You can identify the *pointer type* because there's an asterisk (`*`) before the variable name and after its type:

```
int main(void)
{
    int i;  // i's type is "int"
    int *p; // p's type is "pointer to an int", or "int-pointer"
}
```

[4] That is, base 16 with digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Hey, so we have here a variable that is a pointer type, and it can point to other `int`s. That is, it can hold the address of other `int`s. We know it points to `int`s, since it's of type `int*` (read ``int-pointer'').

When you do an assignment into a pointer variable, the type of the right hand side of the assignment has to be the same type as the pointer variable. Fortunately for us, when you take the `address-of` a variable, the resultant type is a pointer to that variable type, so assignments like the following are perfect:

```
int i;
int *p;  // p is a pointer, but is uninitialized and points to garbage

p = &i;  // p is assigned the address of i--p now "points to" i
```

On the left of the assignment, we have a variable of type pointer-to-`int` (`int*`), and on the right side, we have expression of type pointer-to-`int` since `i` is an `int` (because address-of `int` gives you a pointer to `int`). The address of a thing can be stored in a pointer to that thing.

Get it? I know it still doesn't quite make much sense since you haven't seen an actual use for the pointer variable, but we're taking small steps here so that no one gets lost. So now, let's introduce you to the anti-address-of operator. It's kind of like what `address-of` would be like in Bizarro World.

5.3 Dereferencing

A pointer variable can be thought of as *referring* to another variable by pointing to it. It's rare you'll hear anyone in C land talking about ``referring'' or ``references'', but I bring it up just so that the name of this operator will make a little more sense.

When you have a pointer to a variable (roughly ``a reference to a variable''), you can use the original variable through the pointer by *dereferencing* the pointer. (You can think of this as ``de-pointering'' the pointer, but no one ever says ``de-pointering''.)

Back to our analogy, this is vaguely like looking at a home address and then going to that house.

Now, what do I mean by ``get access to the original variable''? Well, if you have a variable called `i`, and you have a pointer to `i` called `p`, you can use the dereferenced pointer `p` *exactly as if it were the original variable `i`*!

You almost have enough knowledge to handle an example. The last tidbit you need to know is actually this: what is the dereference operator? It's actually called the *indirection operator*, because you're accessing values indirectly via the pointer. And it is the asterisk, again: `*`. Now, don't get this confused with the asterisk you used in the pointer declaration, earlier. They are the same character, but they have different meanings in different contexts[5].

Here's a full-blown example:

```
#include <stdio.h>

int main(void)
{
    int i;
    int *p;  // this is NOT a dereference--this is a type "int*"

    p = &i;  // p now points to i, p holds address of i

    i = 10;  // i is now 10
    *p = 20; // the thing p points to (namely i!) is now 20!!

```

[5]That's not all! It's used in `/*comments*/` and multiplication and in function prototypes with variable length arrays! It's all the same `*`, but the context gives it different meaning.

```
13      printf("i is %d\n", i);   // prints "20"
14      printf("i is %d\n", *p);  // "20"! dereference-p is the same as i!
15  }
```

Remember that `p` holds the address of `i`, as you can see where we did the assignment to `p` on line 8. What the indirection operator does is tells the computer to *use the object the pointer points to* instead of using the pointer itself. In this way, we have turned `*p` into an alias of sorts for `i`.

Great, but *why*? Why do any of this?

5.4 Passing Pointers as Arguments

Right about now, you're thinking that you have an awful lot of knowledge about pointers, but absolutely zero application, right? I mean, what use is `*p` if you could just simply say `i` instead?

Well, my friend, the real power of pointers comes into play when you start passing them to functions. Why is this a big deal? You might recall from before that you could pass all kinds of arguments to functions and they'd be dutifully copied into parameters, and then you could manipulate local copies of those variables from within the function, and then you could return a single value.

What if you wanted to bring back more than one single piece of data from the function? I mean, you can only return one thing, right? What if I answered that question with another question? …Er, two questions?

What happens when you pass a pointer as an argument to a function? Does a copy of the pointer get put into its corresponding parameter? *You bet your sweet peas it does.* Remember how earlier I rambled on and on about how *EVERY SINGLE ARGUMENT* gets copied into parameters and the function uses a copy of the argument? Well, the same is true here. The function will get a copy of the pointer.

But, and this is the clever part: we will have set up the pointer in advance to point at a variable… and then the function can dereference its copy of the pointer to get back to the original variable! The function can't see the variable itself, but it can certainly dereference a pointer to that variable!

This is analogous to writing a home address on a piece of paper, and then copying that onto another piece of paper. You now have *two* pointers to that house, and both are equally good at getting you to the house itself.

In the case of a function call. one of the copies is stored in a pointer variable out in the calling scope, and the other is stored in a pointer variable that is the parameter of the function.

Example! Let's revisit our old `increment()` function, but this time let's make it so that it actually increments the value out in the caller.

```
1   #include <stdio.h>
2
3   void increment(int *p)  // note that it accepts a pointer to an int
4   {
5       *p = *p + 1;        // add one to the thing p points to
6   }
7
8   int main(void)
9   {
10      int i = 10;
11      int *j = &i;  // note the address-of; turns it into a pointer to i
12
13      printf("i is %d\n", i);        // prints "10"
14      printf("i is also %d\n", *j);  // prints "10"
15
16      increment(j);                  // j is an int*--to i
```

```
17
18      printf("i is %d\n", i);        // prints "11"!
19  }
```

Ok! There are a couple things to see here… not the least of which is that the `increment()` function takes an `int*` as an argument. We pass it an `int*` in the call by changing the `int` variable `i` to an `int*` using the `address-of` operator. (Remember, a pointer holds an address, so we make pointers to variables by running them through the `address-of` operator.)

The `increment()` function gets a copy of the pointer. Both the original pointer `j` (in `main()`) and the copy of that pointer `p` (the parameter in `increment()`) point to the same address, namely the one holding the value `i`. (Again, by analogy, like two pieces of paper with the same home address written on them.) Dereferencing either will allow you to modify the original variable `i`! The function can modify a variable in another scope! Rock on!

The above example is often more concisely written in the call just by using address-of right in the argument list:

```
printf("i is %d\n", i);  // prints "10"
increment(&i);
printf("i is %d\n", i);  // prints "11"!
```

Pointer enthusiasts will recall from early on in the guide, we used a function to read from the keyboard, `scanf()`… and, although you might not have recognized it at the time, we used the `address-of` to pass a pointer to a value to `scanf()`. We had to pass a pointer, see, because `scanf()` reads from the keyboard (typically) and stores the result in a variable. The only way it can see that variable out in the calling function's scope is if we pass a pointer to that variable:

```
int i = 0;

scanf("%d", &i);          // pretend you typed "12"
printf("i is %d\n", i);  // prints "i is 12"
```

See, `scanf()` dereferences the pointer we pass it in order to modify the variable it points to. And now you know why you have to put that pesky ampersand in there!

5.5 The `NULL` Pointer

Any pointer variable of any pointer type can be set to a special value called `NULL`. This indicates that this pointer doesn't point to anything.

```
int *p;

p = NULL;
```

Since it doesn't point to a value, dereferencing it is undefined behavior, and probably will result in a crash:

```
int *p = NULL;

*p = 12;  // CRASH or SOMETHING PROBABLY BAD. BEST AVOIDED.
```

Despite being called the billion dollar mistake by its creator[6], the `NULL` pointer is a good sentinel value[7] and general indicator that a pointer hasn't yet been initialized.

(Of course, like other variables, the pointer points to garbage unless you explicitly assign it to point to an address or `NULL`.)

[6] https://en.wikipedia.org/wiki/Null_pointer#History
[7] https://en.wikipedia.org/wiki/Sentinel_value

5.6 A Note on Declaring Pointers

The syntax for declaring a pointer can get a little weird. Let's look at this example:

```
int a;
int b;
```

We can condense that into a single line, right?

```
int a, b;  // Same thing
```

So `a` and `b` are both `int`s. No problem.

But what about this?

```
int a;
int *p;
```

Can we make that into one line? We can. But where does the `*` go?

The rule is that the `*` goes in front of any variable that is a pointer type. That is. the `*` is *not* part of the `int` in this example. it's a part of variable `p`.

With that in mind, we can write this:

```
int a, *p;  // Same thing
```

It's important to note that the following line does *not* declare two pointers:

```
int *p, q;  // p is a pointer to an int; q is just an int.
```

This can be particularly insidious-looking if the programmer writes this following (valid) line of code which is functionally identical to the one above.

```
int* p, q;  // p is a pointer to an int; q is just an int.
```

So take a look at this and determine which variables are pointers and which are not:

```
int *a, b, c, *d, e, *f, g, h, *i;
```

I'll drop the answer in a footnote[8].

5.7 `sizeof` and Pointers

Just a little bit of syntax here that might be confusing and you might see from time to time.

Recall that `sizeof` operates on the *type* of the expression.

```
int *p;

// Prints size of an 'int'
printf("%zu\n", sizeof(int));

// p is type 'int *', so prints size of 'int*'
printf("%zu\n", sizeof p);

// *p is type 'int', so prints size of 'int'
printf("%zu\n", sizeof *p);
```

You might see code in the wild with that last `sizeof` in there. Just remember that `sizeof` is all about the type of the expression, not the variables in the expression themselves.

[8] The pointer type variables are `a`, `d`, `f`, and `i`, because those are the ones with `*` in front of them.

Chapter 6

Arrays

> *``Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."*
>
> ---Stan Kelly-Bootle, computer scientist

Luckily, C has arrays. I mean, I know it's considered a low-level language[1] but it does at least have the concept of arrays built-in. And since a great many languages drew inspiration from C's syntax, you're probably already familiar with using `[` and `]` for declaring and using arrays.

But C only *barely* has arrays! As we'll find out later, arrays are just syntactic sugar in C---they're actually all pointers and stuff deep down. *Freak out!* But for now, let's just use them as arrays. *Phew*.

6.1 Easy Example

Let's just crank out an example:

```
#include <stdio.h>

int main(void)
{
    int i;
    float f[4];  // Declare an array of 4 floats

    f[0] = 3.14159;  // Indexing starts at 0, of course.
    f[1] = 1.41421;
    f[2] = 1.61803;
    f[3] = 2.71828;

    // Print them all out:

    for (i = 0; i < 4; i++) {
        printf("%f\n", f[i]);
    }
}
```

When you declare an array, you have to give it a size. And the size has to be fixed[2].

[1] These days, anyway.

[2] Again, not really, but variable-length arrays---of which I'm not really a fan---are a story for another time.

In the above example, we made an array of 4 `floats`. The value in the square brackets in the declaration lets us know that.

Later on in subsequent lines, we access the values in the array, setting them or getting them, again with square brackets.

Hopefully this looks familiar from languages you already know!

6.2 Getting the Length of an Array

You can't…ish. C doesn't record this information[3]. You have to manage it separately in another variable.

When I say ``can't'', I actually mean there are some circumstances when you *can*. There is a trick to get the number of elements in an array in the scope in which an array is declared. But, generally speaking, this won't work the way you want if you pass the array to a function[4].

Let's take a look at this trick. The basic idea is that you take the `sizeof` the array, and then divide that by the size of each element to get the length. For example, if an `int` is 4 bytes, and the array is 32 bytes long, there must be room for $\frac{32}{4}$ or 8 `int`s in there.

```
int x[12];  // 12 ints

printf("%zu\n", sizeof x);     // 48 total bytes
printf("%zu\n", sizeof(int));  // 4 bytes per int

printf("%zu\n", sizeof x / sizeof(int));  // 48/4 = 12 ints!
```

If it's an array of `chars`, then `sizeof` the array *is* the number of elements, since `sizeof(char)` is defined to be 1. For anything else, you have to divide by the size of each element.

But this trick only works in the scope in which the array was defined. If you pass the array to a function, it doesn't work. Even if you make it ``big'' in the function signature:

```
void foo(int x[12])
{
    printf("%zu\n", sizeof x);     // 8?! What happened to 48?
    printf("%zu\n", sizeof(int));  // 4 bytes per int

    printf("%zu\n", sizeof x / sizeof(int));  // 8/4 = 2 ints?? WRONG.
}
```

This is because when you ``pass'' arrays to functions, you're only passing a pointer to the first element, and that's what `sizeof` measures. More on this in the Passing Single Dimensional Arrays to Functions section, below.

One more thing you can do with `sizeof` and arrays is get the size of an array of a fixed number of elements without declaring the array. This is like how you can get the size of an `int` with `sizeof(int)`.

For example, to see how many bytes would be needed for an array of 48 `doubles`, you can do this:

```
sizeof(double [48]);
```

6.3 Array Initializers

You can initialize an array with constants ahead of time:

[3] Since arrays are just pointers to the first element of the array under the hood, there's no additional information recording the length.

[4] Because when you pass an array to a function, you're actually just passing a pointer to the first element of that array, not the ``entire'' array.

```c
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};  // Initialize with these values

    for (i = 0; i < 5; i++) {
        printf("%d\n", a[i]);
    }
}
```

You should never have more items in your initializer than there is room for in the array, or the compiler will get cranky:

```
foo.c: In function ‘main’:
foo.c:6:39: warning: excess elements in array initializer
    6 |     int a[5] = {22, 37, 3490, 18, 95, 999};
      |                                       ^~~
foo.c:6:39: note: (near initialization for ‘a’)
```

But (fun fact!) you can have *fewer* items in your initializer than there is room for in the array. The remaining elements in the array will be automatically initialized with zero. This is true in general for all types of array initializers: if you have an initializer, anything not explicitly set to a value will be set to zero.

```c
int a[5] = {22, 37, 3490};

// is the same as:

int a[5] = {22, 37, 3490, 0, 0};
```

It's a common shortcut to see this in an initializer when you want to set an entire array to zero:

```c
int a[100] = {0};
```

Which means, ``Make the first element zero, and then automatically make the rest zero, as well."

You can set specific array elements in the initializer, as well, by specifying an index for the value! When you do this, C will happily keep initializing subsequent values for you until the initializer runs out, filling everything else with `0`.

To do this, put the index in square brackets with an = after, and then set the value.

Here's an example where we build an array:

```c
int a[10] = {0, 11, 22, [5]=55, 66, 77};
```

Because we listed index 5 as the start for `55`, the resulting data in the array is:

```
0 11 22 0 0 55 66 77 0 0
```

You can put simple constant expressions in there, as well.

```c
#define COUNT 5

int a[COUNT] = {[COUNT-3]=3, 2, 1};
```

which gives us:

```
0 0 3 2 1
```

Lastly, you can also have C compute the size of the array from the initializer, just by leaving the size off:

```
int a[3] = {22, 37, 3490};

// is the same as:

int a[] = {22, 37, 3490};  // Left the size off!
```

6.4 Out of Bounds!

C doesn't stop you from accessing arrays out of bounds. It might not even warn you.

Let's steal the example from above and keep printing off the end of the array. It only has 5 elements, but let's try to print 10 and see what happens:

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};

    for (i = 0; i < 10; i++) {  // BAD NEWS: printing too many elements!
        printf("%d\n", a[i]);
    }
}
```

Running it on my computer prints:

```
22
37
3490
18
95
32765
1847052032
1780534144
-56487472
21890
```

Yikes! What's that? Well, turns out printing off the end of an array results in what C developers call *undefined behavior*. We'll talk more about this beast later, but for now it means, ``You've done something bad, and anything could happen during your program run."

And by anything, I mean typically things like finding zeroes, finding garbage numbers, or crashing. But really the C spec says in this circumstance the compiler is allowed to emit code that does *anything*[5].

Short version: don't do anything that causes undefined behavior. Ever[6].

[5] In the good old MS-DOS days before memory protection was a thing, I was writing some particularly abusive C code that deliberately engaged in all kinds of undefined behavior. But I knew what I was doing, and things were working pretty well. Until I made a misstep that caused a lockup and, as I found upon reboot, nuked all my BIOS settings. That was fun. (Shout-out to @man for those fun times.)

[6] There are a lot of things that cause undefined behavior, not just out-of-bounds array accesses. This is what makes the C language so *exciting*.

6.5 Multidimensional Arrays

You can add as many dimensions as you want to your arrays.

```
int a[10];
int b[2][7];
int c[4][5][6];
```

These are stored in memory in row-major order[7]. This means with a 2D array, the first index listed indicates the row, and the second the column.

You can also use initializers on multidimensional arrays by nesting them:

```
#include <stdio.h>

int main(void)
{
    int row, col;

    int a[2][5] = {      // Initialize a 2D array
        {0, 1, 2, 3, 4},
        {5, 6, 7, 8, 9}
    };

    for (row = 0; row < 2; row++) {
        for (col = 0; col < 5; col++) {
            printf("(%d,%d) = %d\n", row, col, a[row][col]);
        }
    }
}
```

For output of:

```
(0,0) = 0
(0,1) = 1
(0,2) = 2
(0,3) = 3
(0,4) = 4
(1,0) = 5
(1,1) = 6
(1,2) = 7
(1,3) = 8
(1,4) = 9
```

And you can initialize with explicit indexes:

```
// Make a 3x3 identity matrix

int a[3][3] = {[0][0]=1, [1][1]=1, [2][2]=1};
```

which builds a 2D array like this:

```
1 0 0
0 1 0
0 0 1
```

[7] https://en.wikipedia.org/wiki/Row-_and_column-major_order

6.6 Arrays and Pointers

[*Casually*] So… I kinda might have mentioned up there that arrays were pointers, deep down? We should take a shallow dive into that now so that things aren't completely confusing. Later on, we'll look at what the real relationship between arrays and pointers is, but for now I just want to look at passing arrays to functions.

6.6.1 Getting a Pointer to an Array

I want to tell you a secret. Generally speaking, when a C programmer talks about a pointer to an array, they're talking about a pointer *to the first element* of the array[8].

So let's get a pointer to the first element of an array.

```
#include <stdio.h>

int main(void)
{
    int a[5] = {11, 22, 33, 44, 55};
    int *p;

    p = &a[0];  // p points to the array
                // Well, to the first element, actually

    printf("%d\n", *p);  // Prints "11"
}
```

This is so common to do in C that the language allows us a shorthand:

```
p = &a[0];  // p points to the array

// is the same as:

p = a;      // p points to the array, but much nicer-looking!
```

Just referring to the array name in isolation is the same as getting a pointer to the first element of the array! We're going to use this extensively in the upcoming examples.

But hold on a second---isn't `p` an `int*`? And `*p` gives us `11`, same as `a[0]`? Yessss. You're starting to get a glimpse of how arrays and pointers are related in C.

6.6.2 Passing Single Dimensional Arrays to Functions

Let's do an example with a single dimensional array. I'm going to write a couple functions that we can pass the array to that do different things.

Prepare for some mind-blowing function signatures!

```
#include <stdio.h>

// Passing as a pointer to the first element
void times2(int *a, int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 2);
}
```

[8] This is technically incorrect, as a pointer to an array and a pointer to the first element of an array have different types. But we can burn that bridge when we get to it.

```
9
10  // Same thing, but using array notation
11  void times3(int a[], int len)
12  {
13      for (int i = 0; i < len; i++)
14          printf("%d\n", a[i] * 3);
15  }
16
17  // Same thing, but using array notation with size
18  void times4(int a[5], int len)
19  {
20      for (int i = 0; i < len; i++)
21          printf("%d\n", a[i] * 4);
22  }
23
24  int main(void)
25  {
26      int x[5] = {11, 22, 33, 44, 55};
27
28      times2(x, 5);
29      times3(x, 5);
30      times4(x, 5);
31  }
```

All those methods of listing the array as a parameter in the function are identical.

```
void times2(int *a, int len)
void times3(int a[], int len)
void times4(int a[5], int len)
```

In usage by C regulars, the first is the most common, by far.

And, in fact, in the latter situation, the compiler doesn't even care what number you pass in (other than it has to be greater than zero[9]). It doesn't enforce anything at all.

Now that I've said that, the size of the array in the function declaration actually *does* matter when you're passing multidimensional arrays into functions, but let's come back to that.

6.6.3 Changing Arrays in Functions

We've said that arrays are just pointers in disguise. This means that if you pass an array to a function, you're likely passing a pointer to the first element in the array.

But if the function has a pointer to the data, it is able to manipulate that data! So changes that a function makes to an array will be visible back out in the caller.

Here's an example where we pass a pointer to an array to a function, the function manipulates the values in that array, and those changes are visible out in the caller.

```
1  #include <stdio.h>
2
3  void double_array(int *a, int len)
4  {
```

[9] C11 §6.7.6.2¶1 requires it be greater than zero. But you might see code out there with arrays declared of zero length at the end of `structs` and GCC is particularly lenient about it unless you compile with `-pedantic`. This zero-length array was a hackish mechanism for making variable-length structures. Unfortunately, it's technically undefined behavior to access such an array even though it basically worked everywhere. C99 codified a well-defined replacement for it called *flexible array members*, which we'll chat about later.

```
    // Multiply each element by 2
    //
    // This doubles the values in x in main() since x and a both point
    // to the same array in memory!

    for (int i = 0; i < len; i++)
        a[i] *= 2;
}

int main(void)
{
    int x[5] = {1, 2, 3, 4, 5};

    double_array(x, 5);

    for (int i = 0; i < 5; i++)
        printf("%d\n", x[i]);  // 2, 4, 6, 8, 10!
}
```

Even though we passed the array in as parameter `a` which is type `int*`, look at how we access it using array notation with `a[i]`! Whaaaat. This is totally allowed.

Later when we talk about the equivalence between arrays and pointers, we'll see how this makes a lot more sense. For now, it's enough to know that functions can make changes to arrays that are visible out in the caller.

6.6.4 Passing Multidimensional Arrays to Functions

The story changes a little when we're talking about multidimensional arrays. C needs to know all the dimensions (except the first one) so it has enough information to know where in memory to look to find a value.

Here's an example where we're explicit with all the dimensions:

```
#include <stdio.h>

void print_2D_array(int a[2][3])
{
    for (int row = 0; row < 2; row++) {
        for (int col = 0; col < 3; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int x[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    print_2D_array(x);
}
```

But in this case, these two[10] are equivalent:

```
void print_2D_array(int a[2][3])
void print_2D_array(int a[][3])
```

The compiler really only needs the second dimension so it can figure out how far in memory to skip for each increment of the first dimension. In general, it needs to know all the dimensions except the first one.

Also, remember that the compiler does minimal compile-time bounds checking (if you're lucky), and C does zero runtime checking of bounds. No seat belts! Don't crash by accessing array elements out of bounds!

[10] This is also equivalent: `void print_2D_array(int (*a)[3])`, but that's more than I want to get into right now.

Chapter 7

Strings

Finally! Strings! What could be simpler?

Well, turns out strings aren't actually strings in C. That's right! They're pointers! Of course they are!

Much like arrays, strings in C *barely exist*.

But let's check it out---it's not really such a big deal.

7.1 String Literals

Before we start, let's talk about string literals in C. These are sequences of characters in *double* quotes (`"`). (Single quotes enclose characters, and are a different animal entirely.)

Examples:

```
"Hello, world!\n"
"This is a test."
"When asked if this string had quotes in it, she replied, \"It does.\""
```

The first one has a newline at the end---quite a common thing to see.

The last one has quotes embedded within it, but you see each is preceded by (we say ``escaped by") a backslash (`\`) indicating that a literal quote belongs in the string at this point. This is how the C compiler can tell the difference between printing a double quote and the double quote at the end of the string.

7.2 String Variables

Now that we know how to make a string literal, let's assign it to a variable so we can do something with it.

```
char *s = "Hello, world!";
```

Check out that type: pointer to a `char`. The string variable `s` is actually a pointer to the first character in that string, namely the `H`.

And we can print it with the `%s` (for ``string") format specifier:

```
char *s = "Hello, world!";

printf("%s\n", s);  // "Hello, world!"
```

7.3 String Variables as Arrays

Another option is this, nearly equivalent to the above `char*` usage:

```
char s[14] = "Hello, world!";

// or, if we were properly lazy and have the compiler
// figure the length for us:

char s[] = "Hello, world!";
```

This means you can use array notation to access characters in a string. Let's do exactly that to print all the characters in a string on the same line:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);
}
```

Note that we're using the format specifier `%c` to print a single character.

Also, check this out. The program will still work fine if we change the definition of `s` to be a `char*` type:

```
#include <stdio.h>

int main(void)
{
    char *s = "Hello, world!";   // char* here

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);    // But still use arrays here...?
}
```

And we still can use array notation to get the job done when printing it out! This is surprising, but is still only because we haven't talked about array/pointer equivalence yet. But this is yet another hint that arrays and pointers are the same thing, deep down.

7.4 String Initializers

We've already seen some examples with initializing string variables with string literals:

```
char *s = "Hello, world!";
char t[] = "Hello, again!";
```

But these two are subtly different.

This one is a pointer to a string literal (i.e. a pointer to the first character in a string):

```
char *s = "Hello, world!";
```

If you try to mutate that string with this:

```
char *s = "Hello, world!";

s[0] = 'z';  // BAD NEWS: tried to mutate a string literal!
```

The behavior is undefined. Probably, depending on your system, a crash will result.

But declaring it as an array is different. This one is a mutable *copy* of the string that we can change at will:

```
char t[] = "Hello, again!";  // t is an array copy of the string
t[0] = 'z'; //  No problem

printf("%s\n", t);  // "zello, again!"
```

So remember: if you have a pointer to a string literal, don't try to change it! And if you use a string in double quotes to initialize an array, that's not actually a string literal.

7.5 Getting String Length

You can't, since C doesn't track it for you. And when I say ``can't", I actually mean ``can"[1]. There's a function in `<string.h>` called `strlen()` that can be used to compute the length of any string in bytes[2].

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

int main(void)
{
    char *s = "Hello, world!";

    printf("The string is %zu bytes long.\n", strlen(s));
}
```

The `strlen()` function returns type `size_t`, which is an integer type so you can use it for integer math. We print `size_t` with `%zu`.

The above program prints:

```
The string is 13 bytes long.
```

Great! So it *is* possible to get the string length!

But… if C doesn't track the length of the string anywhere, how does it know how long the string is?

7.6 String Termination

C does strings a little differently than many programming languages, and in fact differently than almost every modern programming language.

When you're making a new language, you have basically two options for storing a string in memory:

1. Store the bytes of the string along with a number indicating the length of the string.
2. Store the bytes of the string, and mark the end of the string with a special byte called the *terminator*.

If you want strings longer than 255 characters, option 1 requires at least two bytes to store the length. Whereas option 2 only requires one byte to terminate the string. So a bit of savings there.

[1] Though it is true that C doesn't track the length of strings.

[2] If you're using the basic character set or an 8-bit character set, you're used to one character being one byte. This isn't true in all character encodings, though.

Of course, these days it seems ridiculous to worry about saving a byte (or 3---lots of languages will happily let you have strings that are 4 gigabytes in length). But back in the day, it was a bigger deal.

So C took approach #2. In C, a ``string'' is defined by two basic characteristics:

- A pointer to the first character in the string.
- A zero-valued byte (or `NUL` character[3]) somewhere in memory after the pointer that indicates the end of the string.

A `NUL` character can be written in C code as `\0`, though you don't often have to do this.

When you include a string in double quotes in your code, the `NUL` character is automatically, implicitly included.

```
char *s = "Hello!";  // Actually "Hello!\0" behind the scenes
```

So with this in mind, let's write our own `strlen()` function that counts `char`s in a string until it finds a `NUL`.

The procedure is to look down the string for a single `NUL` character, counting as we go[4]:

```
int my_strlen(char *s)
{
    int count = 0;

    while (s[count] != '\0')  // Single quotes for single char
        count++;

    return count;
}
```

And that's basically how the built-in `strlen()` gets the job done.

7.7 Copying a String

You can't copy a string through the assignment operator (=). All that does is make a copy of the pointer to the first character… so you end up with two pointers to the same string:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";
    char *t;

    // This makes a copy of the pointer, not a copy of the string!
    t = s;

    // We modify t
    t[0] = 'z';

    // But printing s shows the modification!
    // Because t and s point to the same string!

    printf("%s\n", s);  // "zello, world!"
}
```

[3] This is different than the `NULL` pointer, and I'll abbreviate it `NUL` when talking about the character versus `NULL` for the pointer.

[4] Later we'll learn a neater way to do it with pointer arithmetic.

If you want to make a copy of a string, you have to copy it a byte at a time---but this is made easier with the `strcpy()` function[5].

Before you copy the string, make sure you have room to copy it into, i.e. the destination array that's going to hold the characters needs to be at least as long as the string you're copying.

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

int main(void)
{
    char s[] = "Hello, world!";
    char t[100];  // Each char is one byte, so plenty of room

    // This makes a copy of the string!
    strcpy(t, s);

    // We modify t
    t[0] = 'z';

    // And s remains unaffected because it's a different string
    printf("%s\n", s);  // "Hello, world!"

    // But t has been changed
    printf("%s\n", t);  // "zello, world!"
}
```

Notice with `strcpy()`, the destination pointer is the first argument, and the source pointer is the second. A mnemonic I use to remember this is that it's the order you would have put `t` and `s` if an assignment = worked for strings, with the source on the right and the destination on the left.

[5]There's a safer function called `strncpy()` that you should probably use instead, but we'll get to that later.

Chapter 8

Structs

In C, we have something called a `struct`, which is a user-definable type that holds multiple pieces of data, potentially of different types.

It's a convenient way to bundle multiple variables into a single one. This can be beneficial for passing variables to functions (so you just have to pass one instead of many), and useful for organizing data and making code more readable.

If you've come from another language, you might be familiar with the idea of *classes* and *objects*. These don't exist in C, natively[1]. You can think of a `struct` as a class with only data members, and no methods.

8.1 Declaring a Struct

You can declare a `struct` in your code like so:

```
struct car {
    char *name;
    float price;
    int speed;
};
```

This is often done at the global scope outside any functions so that the `struct` is globally available.

When you do this, you're making a new *type*. The full type name is `struct car`. (Not just `car`---that won't work.)

There aren't any variables of that type yet, but we can declare some:

```
struct car saturn;  // Variable "saturn" of type "struct car"
```

And now we have an uninitialized variable `saturn`[2] of type `struct car`.

We should initialize it! But how do we set the values of those individual fields?

Like in many other languages that stole it from C, we're going to use the dot operator (`.`) to access the individual fields.

```
saturn.name = "Saturn SL/2";
saturn.price = 15999.99;
```

[1] Although in C individual items in memory like `int`s are referred to as ``objects", they're not objects in an object-oriented programming sense.

[2] The Saturn was a popular brand of economy car in the United States until it was put out of business by the 2008 crash, sadly so to us fans.

```
saturn.speed = 175;

printf("Name:           %s\n", saturn.name);
printf("Price (USD):    %f\n", saturn.price);
printf("Top Speed (km): %d\n", saturn.speed);
```

There on the first lines, we set the values in the `struct car`, and then in the next bit, we print those values out.

8.2 Struct Initializers

That example in the previous section was a little unwieldy. There must be a better way to initialize that `struct` variable!

You can do it with an initializer by putting values in for the fields *in the order they appear in the `struct`* when you define the variable. (This won't work after the variable has been defined---it has to happen in the definition).

```
struct car {
    char *name;
    float price;
    int speed;
};

// Now with an initializer! Same field order as in the struct declaration:
struct car saturn = {"Saturn SL/2", 16000.99, 175};

printf("Name:      %s\n", saturn.name);
printf("Price:     %f\n", saturn.price);
printf("Top Speed: %d km\n", saturn.speed);
```

The fact that the fields in the initializer need to be in the same order is a little freaky. If someone changes the order in `struct car`, it could break all the other code!

We can be more specific with our initializers:

```
struct car saturn = {.speed=175, .name="Saturn SL/2"};
```

Now it's independent of the order in the `struct` declaration. Which is safer code, for sure.

Similar to array initializers, any missing field designators are initialized to zero (in this case, that would be `.price`, which I've omitted).

8.3 Passing Structs to Functions

You can do a couple things to pass a `struct` to a function.

1. Pass the `struct`.
2. Pass a pointer to the `struct`.

Recall that when you pass something to a function, a *copy* of that thing gets made for the function to operate on, whether it's a copy of a pointer, an `int`, a `struct`, or anything.

There are basically two cases when you'd want to pass a pointer to the `struct`:

1. You need the function to be able to make changes to the `struct` that was passed in, and have those changes show in the caller.

2. The `struct` is somewhat large and it's more expensive to copy that onto the stack than it is to just copy a pointer[3].

For those two reasons, it's far more common to pass a pointer to a `struct` to a function, though its by no means illegal to pass the `struct` itself.

Let's try passing in a pointer, making a function that will allow you to set the `.price` field of the `struct car`:

```
#include <stdio.h>

struct car {
    char *name;
    float price;
    int speed;
};

int main(void)
{
    struct car saturn = {.speed=175, .name="Saturn SL/2"};

    // Pass a pointer to this struct car, along with a new,
    // more realistic, price:
    set_price(&saturn, 799.99);

    printf("Price: %f\n", saturn.price);
}
```

You should be able to come up with the function signature for `set_price()` just by looking at the types of the arguments we have there.

`saturn` is a `struct car`, so `&saturn` must be the address of the `struct car`, AKA a pointer to a `struct car`, namely a `struct car*`.

And `799.99` is a `float`.

So the function declaration must look like this:

```
void set_price(struct car *c, float new_price)
```

We just need to write the body. One attempt might be:

```
void set_price(struct car *c, float new_price) {
    c.price = new_price;  // ERROR!!
}
```

That won't work because the dot operator only works on `structs`... it doesn't work on *pointers* to `structs`.

Ok, so we can dereference the `struct` to de-pointer it to get to the `struct` itself. Dereferencing a `struct car*` results in the `struct car` that the pointer points to, which we should be able to use the dot operator on:

```
void set_price(struct car *c, float new_price) {
    (*c).price = new_price;  // Works, but is ugly and non-idiomatic :(
}
```

And that works! But it's a little clunky to type all those parens and the asterisk. C has some syntactic sugar called the *arrow operator* that helps with that.

[3] A pointer is likely 8 bytes on a 64-bit system.

8.4 The Arrow Operator

The arrow operator helps refer to fields in pointers to `structs`.

```
void set_price(struct car *c, float new_price) {
    // (*c).price = new_price;  // Works, but non-idiomatic :(
    //
    // The line above is 100% equivalent to the one below:

    c->price = new_price;  // That's the one!
}
```

So when accessing fields, when do we use dot and when do we use arrow?

- If you have a `struct`, use dot (`.`).
- If you have a pointer to a `struct`, use arrow (`->`).

8.5 Copying and Returning `structs`

Here's an easy one for you!

Just assign from one to the other!

```
struct car a, b;

b = a;  // Copy the struct
```

And returning a `struct` (as opposed to a pointer to one) from a function also makes a similar copy to the receiving variable.

This is not a ``deep copy''[4]. All fields are copied as-is, including pointers to things.

8.6 Comparing `structs`

There's only one safe way to do it: compare each field one at a time.

You might think you could use `memcmp()`[5], but that doesn't handle the case of the possible padding bytes that might be in there.

If you clear the `struct` to zero first with `memset()`[6], then it *might* work, though there could be weird elements that might not compare as you expect[7].

[4] A *deep copy* follows pointer in the `struct` and copies the data they point to, as well. A *shallow copy* just copies the pointers, but not the things they point to. C doesn't come with any built-in deep copy functionality.

[5] https://beej.us/guide/bgclr/html/split/stringref.html#man-strcmp

[6] https://beej.us/guide/bgclr/html/split/stringref.html#man-memset

[7] https://stackoverflow.com/questions/141720/how-do-you-compare-structs-for-equality-in-c

Chapter 9

File Input/Output

We've already seen a couple examples of I/O with `scanf()` and `printf()` for doing I/O at the console (screen/keyboard).

But we'll push those concepts a little farther this chapter.

9.1 The `FILE*` Data Type

When we do any kind of I/O in C, we do so through a piece of data that you get in the form of a `FILE*` type. This `FILE*` holds all the information needed to communicate with the I/O subsystem about which file you have open, where you are in the file, and so on.

The spec refers to these as *streams*, i.e. a stream of data from a file or from any source. I'm going to use ``files" and ``streams" interchangeably, but really you should think of a ``file" as a special case of a ``stream". There are other ways to stream data into a program than just reading from a file.

We'll see in a moment how to go from having a filename to getting an open `FILE*` for it, but first I want to mention three streams that are already open for you and ready for use.

`FILE*` name	Description
`stdin`	Standard Input, generally the keyboard by default
`stdout`	Standard Output, generally the screen by default
`stderr`	Standard Error, generally the screen by default, as well

We've actually been using these implicitly already, it turns out. For example, these two calls are the same:

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n");  // printf to a file
```

But more on that later.

Also you'll notice that both `stdout` and `stderr` go to the screen. While this seems at first either like an oversight or redundancy, it actually isn't. Typical operating systems allow you to *redirect* the output of either of those into different files, and it can be convenient to be able to separate error messages from regular non-error output.

For example, in a POSIX shell (like sh, ksh, bash, zsh, etc.) on a Unix-like system, we could run a program and send just the non-error (`stdout`) output to one file, and all the error (`stderr`) output to another file.

```
./foo > output.txt 2> errors.txt   # This command is Unix-specific
```

For this reason, you should send serious error messages to `stderr` instead of `stdout`.

More on how to do that later.

9.2 Reading Text Files

Streams are largely categorized two different ways: *text* and *binary*.

Text streams are allowed to do significant translation of the data, most notably translations of newlines to their different representations[1]. Text files are logically a sequence of *lines* separated by newlines. To be portable, your input data should always end with a newline.

But the general rule is that if you're able to edit the file in a regular text editor, it's a text file. Otherwise, it's binary. More on binary later.

So let's get to work---how do we open a file for reading, and pull data out of it?

Let's create a file called `hello.txt` that has just this in it:

```
Hello, world!
```

And let's write a program to open the file, read a character out of it, and then close the file when we're done. That's the game plan!

```
#include <stdio.h>

int main(void)
{
    FILE *fp;                   // Variable to represent open file

    fp = fopen("hello.txt", "r");  // Open file for reading

    int c = fgetc(fp);          // Read a single character
    printf("%c\n", c);          // Print char to stdout

    fclose(fp);                 // Close the file when done
}
```

See how when we opened the file with `fopen()`, it returned the `FILE*` to us so we could use it later.

(I'm leaving it out for brevity, but `fopen()` will return `NULL` if something goes wrong, like file-not-found, so you should really error check it!)

Also notice the `"r"` that we passed in---this means ``open a text stream for reading''. (There are various strings we can pass to `fopen()` with additional meaning, like writing, or appending, and so on.)

After that, we used the `fgetc()` function to get a character from the stream. You might be wondering why I've made `c` an `int` instead of a `char`---hold that thought!

Finally, we close the stream when we're done with it. All streams are automatically closed when the program exits, but it's good form and good housekeeping to explicitly close any files yourself when done with them.

The `FILE*` keeps track of our position in the file. So subsequent calls to `fgetc()` would get the next character in the file, and then the next, until the end.

[1]We used to have three different newlines in broad effect: Carriage Return (CR, used on old Macs), Linefeed (LF, used on Unix systems), and Carriage Return/Linefeed (CRLF, used on Windows systems). Thankfully the introduction of OS X, being Unix-based, reduced this number to two.

But that sounds like a pain. Let's see if we can make it easier.

9.3 End of File: `EOF`

There is a special character defined as a macro: `EOF`. This is what `fgetc()` will return when the end of the file has been reached and you've attempted to read another character.

How about I share that Fun Fact™, now. Turns out `EOF` is the reason why `fgetc()` and functions like it return an `int` instead of a `char`. `EOF` isn't a character proper, and its value likely falls outside the range of `char`. Since `fgetc()` needs to be able to return any byte **and** `EOF`, it needs to be a wider type that can hold more values. so `int` it is. But unless you're comparing the returned value against `EOF`, you can know, deep down, it's a `char`.

All right! Back to reality! We can use this to read the whole file in a loop.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp;
6      int c;
7
8      fp = fopen("hello.txt", "r");
9
10     while ((c = fgetc(fp)) != EOF)
11         printf("%c", c);
12
13     fclose(fp);
14 }
```

(If line 10 is too weird, just break it down starting with the innermost-nested parens. The first thing we do is assign the result of `fgetc()` into `c`, and *then* we compare *that* against `EOF`. We've just crammed it into a single line. This might look hard to read, but study it---it's idiomatic C.)

And running this, we see:

```
Hello, world!
```

But still, we're operating a character at a time, and lots of text files make more sense at the line level. Let's switch to that.

9.3.1 Reading a Line at a Time

So how can we get an entire line at once? `fgets()` to the rescue! For arguments, it takes a pointer to a `char` buffer to hold bytes, a maximum number of bytes to read, and a `FILE*` to read from. It returns `NULL` on end-of-file or error. `fgets()` is even nice enough to NUL-terminate the string when its done[2].

Let's do a similar loop as before, except let's have a multiline file and read it in a line at a time.

Here's a file `quote.txt`:

```
A wise man can learn more from
a foolish question than a fool
can learn from a wise answer.
                  --Bruce Lee
```

[2] If the buffer's not big enough to read in an entire line, it'll just stop reading mid-line, and the next call to `fgets()` will continue reading the rest of the line.

And here's some code that reads that file a line at a time and prints out a line number before each one:

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char s[1024];  // Big enough for any line this program will encounter
    int linecount = 0;

    fp = fopen("quote.txt", "r");

    while (fgets(s, sizeof s, fp) != NULL)
        printf("%d: %s", ++linecount, s);

    fclose(fp);
}
```

Which gives the output:

```
1: A wise man can learn more from
2: a foolish question than a fool
3: can learn from a wise answer.
4:                   --Bruce Lee
```

9.4 Formatted Input

You know how you can get formatted output with `printf()` (and, thus, `fprintf()` like we'll see, below)?

You can do the same thing with `fscanf()`.

Let's have a file with a series of data records in it. In this case, whales, with name, length in meters, and weight in tonnes. `whales.txt`:

```
blue 29.9 173
right 20.7 135
gray 14.9 41
humpback 16.0 30
```

Yes, we could read these with `fgets()` and then parse the string with `sscanf()` (and in some ways that's more resilient against corrupted files), but in this case, let's just use `fscanf()` and pull it in directly.

The `fscanf()` function skips leading whitespace when reading, and returns `EOF` on end-of-file or error.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char name[1024];  // Big enough for any line this program will encounter
    float length;
    int mass;

    fp = fopen("whales.txt", "r");

    while (fscanf(fp, "%s %f %d", name, &length, &mass) != EOF)
        printf("%s whale, %d tonnes, %.1f meters\n", name, mass, length);
```

```
14
15      fclose(fp);
16  }
```

Which gives the result:

```
blue whale, 173 tonnes, 29.9 meters
right whale, 135 tonnes, 20.7 meters
gray whale, 41 tonnes, 14.9 meters
humpback whale, 30 tonnes, 16.0 meters
```

9.5 Writing Text Files

In much the same way we can use `fgetc()`, `fgets()`, and `fscanf()` to read text streams, we can use `fputc()`, `fputs()`, and `fprintf()` to write text streams.

To do so, we have to `fopen()` the file in write mode by passing `"w"` as the second argument. Opening an existing file in `"w"` mode will instantly truncate that file to 0 bytes for a full overwrite.

We'll put together a simple program that outputs a file `output.txt` using a variety of output functions.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       FILE *fp;
6       int x = 32;
7
8       fp = fopen("output.txt", "w");
9
10      fputc('B', fp);
11      fputc('\n', fp);   // newline
12      fprintf(fp, "x = %d\n", x);
13      fputs("Hello, world!\n", fp);
14
15      fclose(fp);
16  }
```

And this produces a file, `output.txt`, with these contents:

```
B
x = 32
Hello, world!
```

Fun fact: since `stdout` is a file, you could replace line 8 with:

```
fp = stdout;
```

and the program would have outputted to the console instead of to a file. Try it!

9.6 Binary File I/O

So far we've just been talking text files. But there's that other beast we mentioned early on called *binary* files, or binary streams.

These work very similarly to text files, except the I/O subsystem doesn't perform any translations on the data like it might with a text file. With binary files, you get a raw stream of bytes, and that's all.

The big difference in opening the file is that you have to add a `"b"` to the mode. That is, to read a binary file, open it in `"rb"` mode. To write a file, open it in `"wb"` mode.

Because it's streams of bytes, and streams of bytes can contain NUL characters, and the NUL character is the end-of-string marker in C, it's rare that people use the `fprintf()`-and-friends functions to operate on binary files.

Instead the most common functions are `fread()` and `fwrite()`. The functions read and write a specified number of bytes to the stream.

To demo, we'll write a couple programs. One will write a sequence of byte values to disk all at once. And the second program will read a byte at a time and print them out[3].

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    unsigned char bytes[6] = {5, 37, 0, 88, 255, 12};

    fp = fopen("output.bin", "wb");  // wb mode for "write binary"!

    // In the call to fwrite, the arguments are:
    //
    // * Pointer to data to write
    // * Size of each "piece" of data
    // * Count of each "piece" of data
    // * FILE*

    fwrite(bytes, sizeof(char), 6, fp);

    fclose(fp);
}
```

Those two middle arguments to `fwrite()` are pretty odd. But basically what we want to tell the function is, ``We have items that are *this* big, and we want to write *that* many of them." This makes it convenient if you have a record of a fixed length, and you have a bunch of them in an array. You can just tell it the size of one record and how many to write.

In the example above, we tell it each record is the size of a `char`, and we have 6 of them.

Running the program gives us a file `output.bin`, but opening it in a text editor doesn't show anything friendly! It's binary data---not text. And random binary data I just made up, at that!

If I run it through a hex dump[4] program, we can see the output as bytes:

```
05 25 00 58 ff 0c
```

And those values in hex do match up to the values (in decimal) that we wrote out.

But now let's try to read them back in with a different program. This one will open the file for binary reading (`"rb"` mode) and will read the bytes one at a time in a loop.

`fread()` has the neat feature where it returns the number of bytes read, or `0` on EOF. So we can loop until we see that, printing numbers as we go.

[3] Normally the second program would read all the bytes at once, and *then* print them out in a loop. That would be more efficient. But we're going for demo value, here.

[4] https://en.wikipedia.org/wiki/Hex_dump

```c
#include <stdio.h>

int main(void)
{
    FILE *fp;
    unsigned char c;

    fp = fopen("output.bin", "rb"); // rb for "read binary"!

    while (fread(&c, sizeof(char), 1, fp) > 0)
        printf("%d\n", c);
}
```

And, running it, we see our original numbers!

```
5
37
0
88
255
12
```

Woo hoo!

9.6.1 `struct` and Number Caveats

As we saw in the `structs` section, the compiler is free to add padding to a `struct` as it sees fit. And different compilers might do this differently. And the same compiler on different architectures could do it differently. And the same compiler on the same architectures could do it differently.

What I'm getting at is this: it's not portable to just `fwrite()` an entire `struct` out to a file when you don't know where the padding will end up.

How do we fix this? Hold that thought---we'll look at some ways to do this after looking at another related problem.

Numbers!

Turns out all architectures don't represent numbers in memory the same way.

Let's look at a simple `fwrite()` of a 2-byte number. We'll write it in hex so each byte is clear. The most significant byte will have the value `0x12` and the least significant will have the value `0x34`.

```c
unsigned short v = 0x1234;  // Two bytes, 0x12 and 0x34

fwrite(&v, sizeof v, 1, fp);
```

What ends up in the stream?

Well, it seems like it should be `0x12` followed by `0x34`, right?

But if I run this on my machine and hex dump the result, I get:

```
34 12
```

They're reversed! What gives?

This has something to do with what's called the *endianess*[5] of the architecture. Some write the most significant bytes first, and some the least significant bytes first.

[5] https://en.wikipedia.org/wiki/Endianess

This means that if you write a multibyte number out straight from memory, you can't do it in a portable way[6].

A similar problem exists with floating point. Most systems use the same format for their floating point numbers, but some do not. No guarantees!

So… how can we fix all these problems with numbers and `structs` to get our data written in a portable way?

The summary is to *serialize* the data, which is a general term that means to take all the data and write it out in a format that you control, that is well-known, and programmable to work the same way on all platforms.

As you might imagine, this is a solved problem. There are a bunch of serialization libraries you can take advantage of, such as Google's *protocol buffers*[7], out there and ready to use. They will take care of all the gritty details for you, and even will allow data from your C programs to interoperate with other languages that support the same serialization methods.

Do yourself and everyone a favor! Serialize your binary data when you write it to a stream! This will keep things nice and portable, even if you transfer data files from one architecture to another.

[6] And this is why I used individual bytes in my `fwrite()` and `fread()` examples, above, shrewdly.
[7] https://en.wikipedia.org/wiki/Protocol_buffers

Chapter 10

`typedef`: Making New Types

Well, not so much making *new* types as getting new names for existing types. Sounds kinda pointless on the surface, but we can really use this to make our code cleaner.

10.1 `typedef` in Theory

Basically, you take an existing type and you make an alias for it with `typedef`.

Like this:

```
typedef int antelope;  // Make "antelope" an alias for "int"

antelope x = 10;       // Type "antelope" is the same as type "int"
```

You can take any existing type and do it. You can even make a number of types with a comma list:

```
typedef int antelope, bagel, mushroom;  // These are all "int"
```

That's really useful, right? That you can type `mushroom` instead of `int`? You must be *super excited* about this feature!

OK, Professor Sarcasm---we'll get to some more common applications of this in a moment.

10.1.1 Scoping

`typedef` follows regular scoping rules.

For this reason, it's quite common to find `typedef` at file scope (``global'') so that all functions can use the new types at will.

10.2 `typedef` in Practice

So renaming `int` to something else isn't that exciting. Let's see where `typedef` commonly makes an appearance.

10.2.1 `typedef` and `structs`

Sometimes a `struct` will be `typedef`'d to a new name so you don't have to type the word `struct` over and over.

```
struct animal {
    char *name;
    int leg_count, speed;
};

//  original name      new name
//            |         |
//            v         v
//      |-----------| |----|
typedef struct animal animal;

struct animal y;  // This works
animal z;         // This also works because "animal" is an alias
```

Personally, I don't care for this practice. I like the clarity the code has when you add the word `struct` to the type; programmers know what they're getting. But it's really common so I'm including it here.

Now I want to run the exact same example in a way that you might commonly see. We're going to put the `struct animal` *in* the `typedef`. You can mash it all together like this:

```
//  original name
//            |
//            v
//      |-----------|
typedef struct animal {
    char *name;
    int leg_count, speed;
} animal;                          // <-- new name

struct animal y;  // This works
animal z;         // This also works because "animal" is an alias
```

That's exactly the same as the previous example, just more concise.

But that's not all! There's another common shortcut that you might see in code using what are called *anonymous structures*[1]. It turns out you don't actually need to name the structure in a variety of places, and with `typedef` is one of them.

Let's do the same example with an anonymous structure:

```
//  Anonymous struct! It has no name!
//         |
//         v
//      |----|
typedef struct {
    char *name;
    int leg_count, speed;
} animal;                          // <-- new name

//struct animal y;  // ERROR: this no longer works--no such struct!
animal z;           // This works because "animal" is an alias
```

As another example, we might find something like this:

```
typedef struct {
    int x, y;
```

[1] We'll talk more about these later.

```
} point;

point p = {.x=20, .y=40};

printf("%d, %d\n", p.x, p.y);  // 20, 40
```

10.2.2 `typedef` and Other Types

It's not that using `typedef` with a simple type like `int` is completely useless… it helps you abstract the types to make it easier to change them later.

For example, if you have `float` all over your code in 100 zillion places, it's going to be painful to change them all to `double` if you find you have to do that later for some reason.

But if you prepared a little with:

```
typedef float app_float;

// and

app_float f1, f2, f3;
```

Then if later you want to change to another type, like `long double`, you just need to change the `typedef`:

```
//        voila!
//      |---------|
typedef long double app_float;

// and no need to change this line:

app_float f1, f2, f3;  // Now these are all long doubles
```

10.2.3 `typedef` and Pointers

You can make a type that is a pointer.

```
typedef int *intptr;

int a = 10;
intptr x = &a;  // "intptr" is type "int*"
```

I really don't like this practice. It hides the fact that `x` is a pointer type because you don't see a `*` in the declaration.

IMHO, it's better to explicitly show that you're declaring a pointer type so that other devs can clearly see it and don't mistake `x` for having a non-pointer type.

But at last count, say, 832,007 people had a different opinion.

10.2.4 `typedef` and Capitalization

I've seen all kinds of capitalization on `typedef`.

```
typedef struct {
    int x, y;
} my_point;          // lower snake case

typedef struct {
```

```
    int x, y;
} MyPoint;          // CamelCase

typedef struct {
    int x, y;
} Mypoint;          // Leading uppercase

typedef struct {
    int x, y;
} MY_POINT;          // UPPER SNAKE CASE
```

The C11 specification doesn't dictate one way or another, and shows examples in all uppercase and all lowercase.

K&R2 uses leading uppercase predominantly, but show some examples in uppercase and snake case (with `_t`).

If you have a style guide in use, stick with it. If you don't, grab one and stick with it.

10.3 Arrays and `typedef`

The syntax is a little weird, and this is rarely seen in my experience, but you can `typedef` an array of some number of items.

```
// Make type five_ints an array of 5 ints
typedef int five_ints[5];

five_ints x = {11, 22, 33, 44, 55};
```

I don't like it because it hides the array nature of the variable, but it's possible to do.

Chapter 11

Pointers II: Arithmetic

Time to get more into it with a number of new pointer topics! If you're not up to speed with pointers, check out the first section in the guide on the matter.

11.1 Pointer Arithmetic

Turns out you can do math on pointers, notably addition and subtraction.

But what does it mean when you do that?

In short, if you have a pointer to a type, adding one to the pointer moves to the next item of that type directly after it in memory.

It's **important** to remember that as we move pointers around and look at different places in memory, we need to make sure that we're always pointing to a valid place in memory before we dereference. If we're off in the weeds and we try to see what's there, the behavior is undefined and a crash is a common result.

This is a little chicken-and-eggy with Array/Pointer Equivalence, below, but we're going to give it a shot, anyway.

11.1.1 Adding to Pointers

First, let's take an array of numbers.

```
int a[5] = {11, 22, 33, 44, 55};
```

Then let's get a pointer to the first element in that array:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // Or "int *p = a;" works just as well
```

Then let's print the value there by dereferencing the pointer:

```
printf("%d\n", *p);  // Prints 11
```

Now let's use pointer arithmetic to print the next element in the array, the one at index 1:

```
printf("%d\n", *(p + 1));  // Prints 22!!
```

What happened there? C knows that `p` is a pointer to an `int`. So it knows the `sizeof` an `int`[1] and it knows to skip that many bytes to get to the next `int` after the first one!

[1] Recall that the `sizeof` operator tells you the size in bytes of an object in memory.

In fact, the prior example could be written these two equivalent ways:

```
printf("%d\n", *p);        // Prints 11
printf("%d\n", *(p + 0));  // Prints 11
```

because adding `0` to a pointer results in the same pointer.

Let's think of the upshot here. We can iterate over elements of an array this way instead of using an array:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // Or "int *p = a;" works just as well

for (int i = 0; i < 5; i++) {
    printf("%d\n", *(p + i));  // Same as p[i]!
}
```

And that works the same as if we used array notation! Oooo! Getting closer to that array/pointer equivalence thing! More on this later in this chapter.

But what's actually happening, here? How does it work?

Remember from early on that memory is like a big array, where a byte is stored at each array index?

And the array index into memory has a few names:

- Index into memory
- Location
- Address
- *Pointer!*

So a point is an index into memory, somewhere.

For a random example, say that a number 3490 was stored at address (``index") 23,237,489,202. If we have an `int` pointer to that 3490, that value of that pointer is 23,237,489,202… because the pointer is the memory address. Different words for the same thing.

And now let's say we have another number, 4096, stored right after the 3490 at address 23,237,489,210 (8 higher than the 3490 because each `int` in this example is 8 bytes long).

If we add `1` to that pointer, it actually jumps ahead `sizeof(int)` bytes to the next `int`. It knows to jump that far ahead because it's an `int` pointer. If it were a `float` pointer, it'd jump `sizeof(float)` bytes ahead to get to the next float!

So you can look at the next `int`, by adding `1` to the pointer, the one after that by adding `2` to the pointer, and so on.

11.1.2 Changing Pointers

We saw how we could add an integer to a pointer in the previous section. This time, let's *modify the pointer, itself*.

You can just add (or subtract) integer values directly to (or from) any pointer!

Let's do that example again, except with a couple changes. First, I'm going to add a `999` to the end of our numbers to act as a sentinel value. This will let us know where the end of the data is.

```
int a[] = {11, 22, 33, 44, 55, 999};  // Add 999 here as a sentinel

int *p = &a[0];  // p points to the 11
```

And we also have `p` pointing to the element at index `0` of `a`, namely `11`, just like before.

Now---let's start *incrementing* `p` so that it points at subsequent elements of the array. We'll do this until `p` points to the `999`; that is, we'll do it until `*p == 999`:

```
while (*p != 999) {       // While the thing p points to isn't 999
    printf("%d\n", *p);   // Print it
    p++;                  // Move p to point to the next int!
}
```

Pretty crazy, right?

When we give it a run, first `p` points to `11`. Then we increment `p`, and it points to `22`, and then again, it points to `33`. And so on, until it points to `999` and we quit.

11.1.3 Subtracting Pointers

You can subtract a value from a pointer to get to earlier address, as well, just like we were adding to them before.

But we can also subtract two pointers to find the difference between them, e.g. we can calculate how many `int`s there are between two `int*`s. The catch is that this only works within a single array[2]---if the pointers point to anything else, you get undefined behavior.

Remember how strings are `char*`s in C? Let's see if we can use this to write another variant of `strlen()` to compute the length of a string that utilizes pointer subtraction.

The idea is that if we have a pointer to the beginning of the string, we can find a pointer to the end of the string by scanning ahead for the `NUL` character.

And if we have a pointer to the beginning of the string, and we computed the pointer to the end of the string, we can just subtract the two pointers to come up with the length!

```
#include <stdio.h>

int my_strlen(char *s)
{
    // Start scanning from the beginning of the string
    char *p = s;

    // Scan until we find the NUL character
    while (*p != '\0')
        p++;

    // Return the difference in pointers
    return p - s;
}

int main(void)
{
    printf("%d\n", my_strlen("Hello, world!"));  // Prints "13"
}
```

Remember that you can only use pointer subtraction between two pointers that point to the same array!

[2]Or string, which is really an array of `char`s. Somewhat peculiarly, you can also have a pointer that references *one past* the end of the array without a problem and still do math on it. You just can't dereference it when it's out there.

11.2 Array/Pointer Equivalence

We're finally ready to talk about this! We've seen plenty of examples of places where we've intermixed array notation, but let's give out the *fundamental formula of array/pointer equivalence*:

```
a[b] == *(a + b)
```

Study that! Those are equivalent and can be used interchangeably!

I've oversimplified a bit, because in my above example `a` and `b` can both be expressions, and we might want a few more parentheses to force order of operations in case the expressions are complex.

The spec is specific, as always, declaring (in C11 §6.5.2.1¶2):

> `E1[E2]` is identical to `(*((E1)+(E2)))`

but that's a little harder to grok. Just make sure you include parentheses if the expressions are complicated so all your math happens in the right order.

This means we can *decide* if we're going to use array or pointer notation for any array or pointer (assuming it points to an element of an array).

Let's use an array and pointer with both array and pointer notation:

```
#include <stdio.h>

int main(void)
{
    int a[] = {11, 22, 33, 44, 55};

    int *p = a;  // p points to the first element of a, 11

    // Print all elements of the array a variety of ways:

    for (int i = 0; i < 5; i++)
        printf("%d\n", a[i]);      // Array notation with a

    for (int i = 0; i < 5; i++)
        printf("%d\n", p[i]);      // Array notation with p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(a + i));  // Pointer notation with a

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p + i));  // Pointer notation with p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p++));    // Moving pointer p
        //printf("%d\n", *(a++));    // Moving array variable a--ERROR!
}
```

So you can see that in general, if you have an array variable, you can use pointer or array notion to access elements. Same with a pointer variable.

The one big difference is that you can *modify* a pointer to point to a different address, but you can't do that with an array variable.

11.2.1 Array/Pointer Equivalence in Function Calls

This is where you'll encounter this concept the most, for sure.

If you have a function that takes a pointer argument, e.g.:

```
int my_strlen(char *s)
```

this means you can pass either an array or a pointer to this function and have it work!

```
char s[] = "Antelopes";
char *t = "Wombats";

printf("%d\n", my_strlen(s));  // Works!
printf("%d\n", my_strlen(t));  // Works, too!
```

And it's also why these two function signatures are equivalent:

```
int my_strlen(char *s)    // Works!
int my_strlen(char s[])   // Works, too!
```

11.3 `void` Pointers

You've already seen the `void` keyword used with functions, but this is an entirely separate, unrelated animal.

Sometimes it's useful to have a pointer to a thing *that you don't know the type of*.

I know. Bear with me just a second.

There are basically two use cases for this.

1. A function is going to operate on something byte-by-byte. For example, `memcpy()` copies bytes of memory from one pointer to another, but those pointers can point to any type. `memcpy()` takes advantage of the fact that if you iterate through `char*`s, you're iterating through the bytes of an object no matter what type the object is. More on this in the Multibyte Values subsection.

2. Another function is calling a function you passed to it (a callback), and it's passing you data. You know the type of the data, but the function calling you doesn't. So it passes you `void*`s---'cause it doesn't know the type---and you convert those to the type you need. The built-in `qsort()`[3] and `bsearch()`[4] use this technique.

Let's look at an example, the built-in `memcpy()` function:

```
void *memcpy(void *s1, void *s2, size_t n);
```

This function copies `n` bytes of memory starting from address `s2` into the memory starting at address `s1`.

But look! `s1` and `s2` are `void*`s! Why? What does it mean? Let's run more examples to see.

For instance, we could copy a string with `memcpy()` (though `strcpy()` is more appropriate for strings):

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

int main(void)
{
    char s[] = "Goats!";
    char t[100];

```

[3] https://beej.us/guide/bgclr/html/split/stdlib.html#man-qsort
[4] https://beej.us/guide/bgclr/html/split/stdlib.html#man-bsearch

```
    memcpy(t, s, 7);  // Copy 7 bytes--including the NUL terminator!

    printf("%s\n", t);  // "Goats!"
}
```

Or we can copy some `int`s:

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

int main(void)
{
    int a[] = {11, 22, 33};
    int b[3];

    memcpy(b, a, 3 * sizeof(int));  // Copy 3 ints of data

    printf("%d\n", b[1]);  // 22
}
```

That one's a little wild---you see what we did there with `memcpy()`? We copied the data from `a` to `b`, but we had to specify how many *bytes* to copy, and an `int` is more than one byte.

OK, then---how many bytes does an `int` take? Answer: depends on the system. But we can tell how many bytes any type takes with the `sizeof` operator.

So there's the answer: an `int` takes `sizeof(int)` bytes of memory to store.

And if we have 3 of them in our array, like we did in that example, the entire space used for the 3 `int`s must be `3 * sizeof(int)`.

(In the string example, earlier, it would have been more technically accurate to copy `7 * sizeof(char)` bytes. But `char`s are always one byte large, by definition, so that just devolves into `7 * 1`.)

We could even copy a `float` or a `struct` with `memcpy()`! (Though this is abusive---we should just use `=` for that):

```
struct antelope my_antelope;
struct antelope my_clone_antelope;

// ...

memcpy(&my_clone_antelope, &my_antelope, sizeof my_antelope);
```

Look at how versatile `memcpy()` is! If you have a pointer to a source and a pointer to a destination, and you have the number of bytes you want to copy, you can copy *any type of data*.

Imagine if we didn't have `void*`. We'd have to write specialized `memcpy()` functions for each type:

```
memcpy_int(int *a, int *b, int count);
memcpy_float(float *a, float *b, int count);
memcpy_double(double *a, double *b, int count);
memcpy_char(char *a, char *b, int count);
memcpy_unsigned_char(unsigned char *a, unsigned char *b, int count);

// etc... blech!
```

Much better to just use `void*` and have one function that can do it all.

That's the power of `void*`. You can write functions that don't care about the type and is still able to do things with it.

But with great power comes great responsibility. Maybe not *that* great in this case, but there are some limits.

1. You cannot do pointer arithmetic on a `void*`. 2. You cannot dereference a `void*`. 3. You cannot use the arrow operator on a `void*`, since it's also a dereference. 4. You cannot use array notation on a `void*`, since it's also a dereference, as well[5].

And if you think about it, these rules make sense. All those operations rely on knowing the `sizeof` the type of data pointed to, and with `void*`, we don't know the size of the data being pointed to---it could be anything!

But wait---if you can't dereference a `void*` what good can it ever do you?

Like with `memcpy()`, it helps you write generic functions that can handle multiple types of data. But the secret is that, deep down, *you convert the `void*` to another type before you use it*!

And conversion is easy: you can just assign into a variable of the desired type[6].

```
char a = 'X';  // A single char

void *p = &a;  // p points to the 'X'
char *q = p;   // q also points to the 'X'

printf("%c\n", *p);  // ERROR--cannot dereference void*!
printf("%c\n", *q);  // Prints "X"
```

Let's write our own `memcpy()` to try this out. We can copy bytes (`char`s), and we know the number of bytes because it's passed in.

```
void *my_memcpy(void *dest, void *src, int byte_count)
{
    // Convert void*s to char*s
    char *s = src, *d = dest;

    // Now that we have char*s, we can dereference and copy them
    while (byte_count--) {
        *d++ = *s++;
    }

    // Most of these functions return the destination, just in case
    // that's useful to the caller.
    return dest;
}
```

Right there at the beginning, we copy the `void*`s into `char*`s so that we can use them as `char*`s. It's as easy as that.

Then some fun in a while loop, where we decrement `byte_count` until it becomes false (`0`). Remember that with post-decrement, the value of the expression is computed (for `while` to use) and *then* the variable is decremented.

And some fun in the copy, where we assign `*d = *s` to copy the byte, but we do it with post-increment so that both `d` and `s` move to the next byte after the assignment is made.

Lastly, most memory and string functions return a copy of a pointer to the destination string just in case the caller wants to use it.

[5]Because remember that array notation is just a dereference and some pointer math, and you can't dereference a `void*`!

[6]You can also *cast* the `void*` to another type, but we haven't gotten to casts yet.

Now that we've done that, I just want to quickly point out that we can use this technique to iterate over the bytes of *any* object in C, `floats`, `structs`, or anything!

Let's run one more real-world example with the built-in `qsort()` routine that can sort *anything* thanks to the magic of `void*`s.

(In the following example, you can ignore the word `const`, which we haven't covered yet.)

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

// The type of structure we're going to sort
struct animal {
    char *name;
    int leg_count;
};

// This is a comparison function called by qsort() to help it determine
// what exactly to sort by. We'll use it to sort an array of struct
// animals by leg_count.
int compar(const void *elem1, const void *elem2)
{
    // We know we're sorting struct animals, so let's make both
    // arguments pointers to struct animals
    const struct animal *animal1 = elem1;
    const struct animal *animal2 = elem2;

    // Return <0 =0 or >0 depending on whatever we want to sort by.

    // Let's sort ascending by leg_count, so we'll return the difference
    // in the leg_counts
    if (animal1->leg_count > animal2->leg_count)
        return 1;

    if (animal1->leg_count < animal2->leg_count)
        return -1;

    return 0;
}

int main(void)
{
    // Let's build an array of 4 struct animals with different
    // characteristics. This array is out of order by leg_count, but
    // we'll sort it in a second.
    struct animal a[4] = {
        {.name="Dog", .leg_count=4},
        {.name="Monkey", .leg_count=2},
        {.name="Antelope", .leg_count=4},
        {.name="Snake", .leg_count=0}
    };

    // Call qsort() to sort the array. qsort() needs to be told exactly
    // what to sort this data by, and we'll do that inside the compar()
    // function.
```

```
48      //
49      // This call is saying: qsort array a, which has 4 elements, and
50      // each element is sizeof(struct animal) bytes big, and this is the
51      // function that will compare any two elements.
52      qsort(a, 4, sizeof(struct animal), compar);
53
54      // Print them all out
55      for (int i = 0; i < 4; i++) {
56          printf("%d: %s\n", a[i].leg_count, a[i].name);
57      }
58  }
```

As long as you give `qsort()` a function that can compare two items that you have in your array to be sorted, it can sort anything. And it does this without needing to have the types of the items hardcoded in there anywhere. `qsort()` just rearranges blocks of bytes based on the results of the `compar()` function you passed in.

Chapter 12

Manual Memory Allocation

This is one of the big areas where C likely diverges from languages you already know: *manual memory management*.

Other languages uses reference counting, garbage collection, or other means to determine when to allocate new memory for some data---and when to deallocate it when no variables refer to it.

And that's nice. It's nice to be able to not worry about it, to just drop all the references to an item and trust that at some point the memory associated with it will be freed.

But C's not like that, entirely.

Of course, in C, some variables are automatically allocated and deallocated when they come into scope and leave scope. We call these automatic variables. They're your average run-of-the-mill block scope ``local'' variables. No problem.

But what if you want something to persist longer than a particular block? This is where manual memory management comes into play.

You can tell C explicitly to allocate for you a certain number of bytes that you can use as you please. And these bytes will remain allocated until you explicitly free that memory[1].

It's important to free the memory you're done with! If you don't, we call that a *memory leak* and your process will continue to reserve that memory until it exits.

If you manually allocated it, you have to manually free it when you're done with it.

So how do we do this? We're going to learn a couple new functions, and make use of the `sizeof` operator to help us learn how many bytes to allocate.

In common C parlance, devs say that automatic local variables are allocated ``on the stack'', and manually-allocated memory is ``on the heap''. The spec doesn't talk about either of those things, but all C devs will know what you're talking about if you bring them up.

All functions we're going to learn in this chapter can be found in `<stdlib.h>`.

12.1 Allocating and Deallocating, `malloc()` and `free()`

The `malloc()` function accepts a number of bytes to allocate, and returns a void pointer to that block of newly-allocated memory.

[1] Or until the program exits, in which case all the memory allocated by it is freed. Asterisk: some systems allow you to allocate memory that persists after a program exits, but it's system dependent, out of scope for this guide, and you'll certainly never do it on accident.

Since it's a `void*`, you can assign it into whatever pointer type you want… normally this will correspond in some way to the number of bytes you're allocating.

So… how many bytes should I allocate? We can use `sizeof` to help with that. If we want to allocate enough room for a single `int`, we can use `sizeof(int)` and pass that to `malloc()`.

After we're done with some allocated memory, we can call `free()` to indicate we're done with that memory and it can be used for something else. As an argument, you pass the same pointer you got from `malloc()` (or a copy of it). It's undefined behavior to use a memory region after you `free()` it.

Let's try. We'll allocate enough memory for an `int`, and then store something there, and the print it.

```
// Allocate space for a single int (sizeof(int) bytes-worth):

int *p = malloc(sizeof(int));

*p = 12;  // Store something there

printf("%d\n", *p);  // Print it: 12

free(p);  // All done with that memory

//*p = 3490;  // ERROR: undefined behavior! Use after free()!
```

Now, in that contrived example, there's really no benefit to it. We could have just used an automatic `int` and it would have worked. But we'll see how the ability to allocate memory this way has its advantages, especially with more complex data structures.

One more thing you'll commonly see takes advantage of the fact that `sizeof` can give you the size of the result type of any constant expression. So you could put a variable name in there, too, and use that. Here's an example of that, just like the previous one:

```
int *p = malloc(sizeof *p);  // *p is an int, so same as sizeof(int)
```

12.2 Error Checking

All the allocation functions return a pointer to the newly-allocated stretch of memory, or `NULL` if the memory cannot be allocated for some reason.

Some OSes like Linux can be configured in such a way that `malloc()` never returns `NULL`, even if you're out of memory. But despite this, you should always code it up with protections in mind.

```
int *x;

x = malloc(sizeof(int) * 10);

if (x == NULL) {
    printf("Error allocating 10 ints\n");
    // do something here to handle it
}
```

Here's a common pattern that you'll see, where we do the assignment and the condition on the same line:

```
int *x;

if ((x = malloc(sizeof(int) * 10)) == NULL)
    printf("Error allocating 10 ints\n");
```

```
    // do something here to handle it
}
```

12.3 Allocating Space for an Array

We've seen how to allocate space for a single thing; now what about for a bunch of them in an array?

In C, an array is a bunch of the same thing back-to-back in a contiguous stretch of memory.

We can allocate a contiguous stretch of memory---we've seen how to do that. If we wanted 3490 bytes of memory, we could just ask for it:

```
char *p = malloc(3490);  // Voila
```

And---indeed!---that's an array of 3490 `chars` (AKA a string!) since each `char` is 1 byte. In other words, `sizeof(char)` is `1`.

Note: there's no initialization done on the newly-allocated memory---it's full of garbage. Clear it with `memset()` if you want to, or see `calloc()`, below.

But we can just multiply the size of the thing we want by the number of elements we want, and then access them using either pointer or array notation. Example!

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

int main(void)
{
    // Allocate space for 10 ints
    int *p = malloc(sizeof(int) * 10);

    // Assign them values 0-45:
    for (int i = 0; i < 10; i++)
        p[i] = i * 5;

    // Print all values 0, 5, 10, 15, ..., 40, 45
    for (int i = 0; i < 10; i++)
        printf("%d\n", p[i]);

    // Free the space
    free(p);
}
```

The key's in that `malloc()` line. If we know each `int` takes `sizeof(int)` bytes to hold it, and we know we want 10 of them, we can just allocate exactly that many bytes with:

```
sizeof(int) * 10
```

And this trick works for every type. Just pass it to `sizeof` and multiply by the size of the array.

12.4 An Alternative: `calloc()`

This is another allocation function that works similarly to `malloc()`, with two key differences:

- Instead of a single argument, you pass the size of one element, and the number of elements you wish to allocate. It's like it's made for allocating arrays.
- It clears the memory to zero.

You still use `free()` to deallocate memory obtained through `calloc()`.

Here's a comparison of `calloc()` and `malloc()`.

```
// Allocate space for 10 ints with calloc(), initialized to 0:
int *p = calloc(10, sizeof(int));

// Allocate space for 10 ints with malloc(), initialized to 0:
int *q = malloc(10 * sizeof(int));
memset(q, 0, 10 * sizeof(int));   // set to 0
```

Again, the result is the same for both except `malloc()` doesn't zero the memory by default.

12.5 Changing Allocated Size with `realloc()`

If you've already allocated 10 `int`s, but later you decide you need 20, what can you do?

One option is to allocate some new space, and then `memcpy()` the memory over… but it turns out that sometimes you don't need to move anything. And there's one function that's just smart enough to do the right thing in all the right circumstances: `realloc()`.

It takes a pointer to some previously-allocted memory (by `malloc()` or `calloc()`) and a new size for the memory region to be.

It then grows or shrinks that memory, and returns a pointer to it. Sometimes it might return the same pointer (if the data didn't have to be copied elsewhere), or it might return a different one (if the data did have to be copied).

Be sure when you call `realloc()`, you specify the number of *bytes* to allocate, and not just the number of array elements! That is:

```
num_floats *= 2;

np = realloc(p, num_floats);  // WRONG: need bytes, not number of elements!

np = realloc(p, num_floats * sizeof(float));  // Better!
```

Let's allocate an array of 20 `float`s, and then change our mind and make it an array of 40.

We're going to assign the return value of `realloc()` into another pointer just to make sure it's not `NULL`. If it's not, then we can reassign it into our original pointer. (If we just assigned the return value directly into the original pointer, we'd lose that pointer if the function returned `NULL` and we'd have no way to get it back.)

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

int main(void)
{
    // Allocate space for 20 floats
    float *p = malloc(sizeof *p * 20);  // sizeof *p same as sizeof(float)

    // Assign them fractional values 0.0-1.0:
    for (int i = 0; i < 20; i++)
        p[i] = i / 20.0;

    // But wait! Let's actually make this an array of 40 elements
    float *new_p = realloc(p, sizeof *p * 40);

```

```
16      // Check to see if we successfully reallocated
17      if (new_p == NULL) {
18          printf("Error reallocing\n");
19          return 1;
20      }
21
22      // If we did, we can just reassign p
23      p = new_p;
24
25      // And assign the new elements values in the range 1.0-2.0
26      for (int i = 20; i < 40; i++)
27          p[i] = 1.0 + (i - 20) / 20.0;
28
29      // Print all values 0.0-2.0 in the 40 elements:
30      for (int i = 0; i < 40; i++)
31          printf("%f\n", p[i]);
32
33      // Free the space
34      free(p);
35  }
```

Notice in there how we took the return value from `realloc()` and reassigned it into the same pointer variable `p` that we passed in. That's pretty common to do.

Also if line 7 is looking weird, with that `sizeof *p` in there, remember that `sizeof` works on the size of the type of the expression. And the type of `*p` is `float`, so that line is equivalent to `sizeof(float)`.

12.5.1 Reading in Lines of Arbitrary Length

I want to demonstrate two things with this full-blown example.

1. Use of `realloc()` to grow a buffer as we read in more data.
2. Use of `realloc()` to shrink the buffer down to the perfect size after we've completed the read.

What we see here is a loop that calls `fgetc()` over and over to append to a buffer until we see that the last character is a newline.

Once it finds the newline, it shrinks the buffer to just the right size and returns it.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   // Read a line of arbitrary size from a file
5   //
6   // Returns a pointer to the line.
7   // Returns NULL on EOF or error.
8   //
9   // It's up to the caller to free() this pointer when done with it.
10  //
11  // Note that this strips the newline from the result. If you need
12  // it in there, probably best to switch this to a do-while.
13
14  char *readline(FILE *fp)
15  {
16      int offset = 0;   // Index next char goes in the buffer
17      int bufsize = 4;  // Preferably power of 2 initial size
```

```
    char *buf;        // The buffer
    int c;            // The character we've read in

    buf = malloc(bufsize);  // Allocate initial buffer

    if (buf == NULL)   // Error check
        return NULL;

    // Main loop--read until newline or EOF
    while (c = fgetc(fp), c != '\n' && c != EOF) {

        // Check if we're out of room in the buffer accounting
        // for the extra byte for the NUL terminator
        if (offset == bufsize - 1) {  // -1 for the NUL terminator
            bufsize *= 2;  // 2x the space

            char *new_buf = realloc(buf, bufsize);

            if (new_buf == NULL) {
                free(buf);   // On error, free and bail
                return NULL;
            }

            buf = new_buf;  // Successful realloc
        }

        buf[offset++] = c;  // Add the byte onto the buffer
    }

    // We hit newline or EOF...

    // If at EOF and we read no bytes, free the buffer and
    // return NULL to indicate we're at EOF:
    if (c == EOF && offset == 0) {
        free(buf);
        return NULL;
    }

    // Shrink to fit
    if (offset < bufsize - 1) {  // If we're short of the end
        char *new_buf = realloc(buf, offset + 1); // +1 for NUL terminator

        // If successful, point buf to new_buf;
        // otherwise we'll just leave buf where it is
        if (new_buf != NULL)
            buf = new_buf;
    }

    // Add the NUL terminator
    buf[offset] = '\0';

    return buf;
}
```

```
71
72  int main(void)
73  {
74      FILE *fp = fopen("foo.txt", "r");
75
76      char *line;
77
78      while ((line = readline(fp)) != NULL) {
79          printf("%s\n", line);
80          free(line);
81      }
82
83      fclose(fp);
84  }
```

When growing memory like this, it's common (though hardly a law) to double the space needed each step just to minimize the number of `realloc()`s that occur.

Finally you might note that `readline()` returns a pointer to a `malloc()`d buffer. As such, it's up to the caller to explicitly `free()` that memory when it's done with it.

12.5.2 `realloc()` with `NULL`

Trivia time! These two lines are equivalent:

```
char *p = malloc(3490);
char *p = realloc(NULL, 3490);
```

That could be convenient if you have some kind of allocation loop and you don't want to special-case the first `malloc()`.

```
int *p = NULL;
int length = 0;

while (!done) {
    // Allocate 10 more ints:
    length += 10;
    p = realloc(p, sizeof *p * length);

    // Do amazing things
    // ...
}
```

In that example, we didn't need an initial `malloc()` since `p` was `NULL` to start.

12.6 Aligned Allocations

You probably aren't going to need to use this.

And I don't want to get too far off in the weeds talking about it right now, but there's this thing called *memory alignment*, which has to do with the memory address (pointer value) being a multiple of a certain number.

For example, a system might require that 16-bit values begin on memory addresses that are multiples of 2. Or that 64-bit values begin on memory addresses that are multiples of 2, 4, or 8, for example. It depends on the CPU.

Some systems require this kind of alignment for fast memory access, or some even for memory access at all.

Now, if you use `malloc()`, `calloc()`, or `realloc()`, C will give you a chunk of memory that's well-aligned for any value at all, even `structs`. Works in all cases.

But there might be times that you know that some data can be aligned at a smaller boundary, or must be aligned at a larger one for some reason. I imagine this is more common with embedded systems programming.

In those cases, you can specify an alignment with `aligned_alloc()`.

The alignment is an integer power of two greater than zero, so `2`, `4`, `8`, `16`, etc. and you give that to `aligned_alloc()` before the number of bytes you're interested in.

The other restriction is that the number of bytes you allocate needs to be a multiple of the alignment. But this might be changing. See C Defect Report 460[2]

Let's do an example, allocating on a 64-byte boundary:

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

int main(void)
{
    // Allocate 256 bytes aligned on a 64-byte boundary
    char *p = aligned_alloc(64, 256);  // 256 == 64 * 4

    // Copy a string in there and print it
    strcpy(p, "Hello, world!");
    printf("%s\n", p);

    // Free the space
    free(p);
}
```

I want to throw a note here about `realloc()` and `aligned_alloc()`. `realloc()` doesn't have any alignment guarantees, so if you need to get some aligned reallocated space, you'll have to do it the hard way with `memcpy()`.

Here's a non-standard `aligned_realloc()` function, if you need it:

```
void *aligned_realloc(void *ptr, size_t old_size, size_t alignment, size_t size)
{
    char *new_ptr = aligned_alloc(alignment, size);

    if (new_ptr == NULL)
        return NULL;

    size_t copy_size = old_size < size? old_size: size;  // get min

    if (ptr != NULL)
        memcpy(new_ptr, ptr, copy_size);

    free(ptr);

    return new_ptr;
}
```

[2] http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_460

Note that it *always* copies data, taking time, while real `realloc()` will avoid that if it can. So this is hardly efficient. Avoid needing to reallocate custom-aligned data.

Chapter 13

Scope

Scope is all about what variables are visible in what contexts.

13.1 Block Scope

This is the scope of almost all the variables devs define. It includes what other languages might call ``function scope'', i.e. variables that are declared inside functions.

The basic rule is that if you've declared a variable in a block delimited by squirrelly braces, the scope of that variable is that block.

If there's a block inside a block, then variables declared in the *inner* block are local to that block, and cannot be seen in the outer scope.

Once a variable's scope ends, that variable can no longer be referenced, and you can consider its value to be gone into the great bit bucket[1] in the sky.

An example with nested scope:

```
#include <stdio.h>

int main(void)
{
    int a = 12;         // Local to outer block, but visible in inner block

    if  (a == 12) {
        int b = 99;     // Local to inner block, not visible in outer block

        printf("%d %d\n", a, b);  // OK: "12 99"
    }

    printf("%d\n", a);  // OK, we're still in a's scope

    printf("%d\n", b);  // ILLEGAL, out of b's scope
}
```

[1] https://en.wikipedia.org/wiki/Bit_bucket

13.1.1 Where To Define Variables

Another fun fact is that you can define variables anywhere in the block, within reason---they have the scope of that block, but cannot be used before they are defined.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 0;
6
7      printf("%d\n", i);     // OK: "0"
8
9      //printf("%d\n", j);   // ILLEGAL--can't use j before it's defined
10
11     int j = 5;
12
13     printf("%d %d\n", i, j);   // OK: "0 5"
14 }
```

Historically, C required all the variables be defined before any code in the block, but this is no longer the case in the C99 standard.

13.1.2 Variable Hiding

If you have a variable named the same thing at an inner scope as one at an outer scope, the one at the inner scope takes precedence as long as you're running in the inner scope. That is, it *hides* the one at outer scope for the duration of its lifetime.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i = 10;
6
7      {
8          int i = 20;
9
10         printf("%d\n", i);  // Inner scope i, 20 (outer i is hidden)
11     }
12
13     printf("%d\n", i);  // Outer scope i, 10
14 }
```

You might have noticed in that example that I just threw a block in there at line 7, not so much as a `for` or `if` statement to kick it off! This is perfectly legal. Sometimes a dev will want to group a bunch of local variables together for a quick computation and will do this, but it's rare to see.

13.2 File Scope

If you define a variable outside of a block, that variable has *file scope*. It's visible in all functions in the file that come after it, and shared between them. (An exception is if a block defines a variable of the same name, it would hide the one at file scope.)

This is closest to what you would consider to be ``global'' scope in another language.

For example:

```c
#include <stdio.h>

int shared = 10;    // File scope! Visible to the whole file after this!

void func1(void)
{
    shared += 100;  // Now shared holds 110
}

void func2(void)
{
    printf("%d\n", shared);  // Prints "110"
}

int main(void)
{
    func1();
    func2();
}
```

Note that if `shared` were declared at the bottom of the file, it wouldn't compile. It has to be declared *before* any functions use it.

There are ways to further modify items at file scope, namely with static and extern, but we'll talk more about those later.

13.3 `for`-loop Scope

I really don't know what to call this, as C11 §6.8.5.3¶1 doesn't give it a proper name. We've done it already a few times in this guide, as well. It's when you declare a variable inside the first clause of a `for`-loop:

```c
for (int i = 0; i < 10; i++)
    printf("%d\n", i);

printf("%d\n", i);  // ILLEGAL--i is only in scope for the for-loop
```

In that example, `i`'s lifetime begins the moment it is defined, and continues for the duration of the loop.

If the loop body is enclosed in a block, the variables defined in the `for`-loop are visible from that inner scope.

Unless, of course, that inner scope hides them. This crazy example prints `999` five times:

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; i++) {
        int i = 999;  // Hides the i in the for-loop scope
        printf("%d\n", i);
    }
}
```

13.4 A Note on Function Scope

The C spec does refer to *function scope*, but it's used exclusively with *labels*, something we haven't discussed yet. More on that another day.

Chapter 14

Types II: Way More Types!

We're used to `char`, `int`, and `float` types, but it's now time to take that stuff to the next level and see what else we have out there in the types department!

14.1 Signed and Unsigned Integers

So far we've used `int` as a *signed* type, that is, a value that can be either negative or positive. But C also has specific *unsigned* integer types that can only hold positive numbers.

These types are prefaced by the keyword `unsigned`.

```
int a;           // signed
signed int a;    // signed
signed a;        // signed, "shorthand" for "int" or "signed int", rare
unsigned int b;  // unsigned
unsigned c;      // unsigned, shorthand for "unsigned int"
```

Why? Why would you decide you only wanted to hold positive numbers?

Answer: you can get larger numbers in an unsigned variable than you can in a signed ones.

But why is that?

You can think of integers being represented by a certain number of *bits*[1]. On my computer, an `int` is represented by 64 bits.

And each permutation of bits that are either `1` or `0` represents a number. We can decide how to divvy up these numbers.

With signed numbers, we use (roughly) half the permutations to represent negative numbers, and the other half to represent positive numbers.

With unsigned, we use *all* the permutations to represent positive numbers.

On my computer with 64-bit `int`s using two's complement[2] to represent unsigned numbers, I have the following limits on integer range:

[1] ``Bit" is short for *binary digit*. Binary is just another way of representing numbers. Instead of digits 0-9 like we're used to, it's digits 0-1.

[2] https://en.wikipedia.org/wiki/Two%27s_complement

Type	Minimum	Maximum
`int`	`-9,223,372,036,854,775,808`	`9,223,372,036,854,775,807`
`unsigned int`	`0`	`18,446,744,073,709,551,615`

Notice that the largest positive `unsigned int` is approximately twice as large as the largest positive `int`. So you can get some flexibility there.

14.2 Character Types

Remember `char`? The type we can use to hold a single character?

```
char c = 'B';

printf("%c\n", c);  // "B"
```

I have a shocker for you: it's actually an integer.

```
char c = 'B';

// Change this from %c to %d:
printf("%d\n", c);  // 66 (!!)
```

Deep down, `char` is just a small `int`, namely an integer that uses just a single byte of space, limiting its range to…

Here the C spec gets just a little funky. It assures us that a `char` is a single byte, i.e. `sizeof(char) == 1`. But then in C11 §3.6¶3 it goes out of its way to say:

> A byte is composed of a contiguous sequence of bits, *the number of which is implementation-defined.*

Wait---what? Some of you might be used to the notion that a byte is 8 bits, right? I mean, that's what it is, right? And the answer is, ``Almost certainly."[3] But C is an old language, and machines back in the day had, shall we say, a more *relaxed* opinion over how many bits were in a byte. And through the years, C has retained this flexibility.

But assuming your bytes in C are 8 bits, like they are for virtually all machines in the world that you'll ever see, the range of a `char` is…

---So before I can tell you, it turns out that `char`s might be signed or unsigned depending on your compiler. Unless you explicitly specify.

In many cases, just having `char` is fine because you don't care about the sign of the data. But if you need signed or unsigned `char`s, you *must* be specific:

```
char a;          // Could be signed or unsigned
signed char b;   // Definitely signed
unsigned char c; // Definitely unsigned
```

OK, now, finally, we can figure out the range of numbers if we assume that a `char` is 8 bits and your system uses the virtually universal two's complement representation for signed and unsigned[4].

So, assuming those constraints, we can finally figure our ranges:

[3] The industry term for a sequence of exactly, indisputably 8 bits is an *octet*.

[4] In general, f you have an n bit two's complement number, the signed range is -2^{n-1} to $2^{n-1}-1$. And the unsigned range is 0 to 2^n-1.

`char` type	Minimum	Maximum
`signed char`	-128	127
`unsigned char`	0	255

And the ranges for `char` are implementation-defined.

Let me get this straight. `char` is actually a number, so can we do math on it?

Yup! Just remember to keep things in the range of a `char`!

```
#include <stdio.h>

int main(void)
{
    char a = 10, b = 20;

    printf("%d\n", a + b);  // 30!
}
```

What about those constant characters in single quotes, like `'B'`? How does that have a numeric value?

The spec is also hand-wavey here, since C isn't designed to run on a single type of underlying system.

But let's just assume for the moment that your character set is based on ASCII[5] for at least the first 128 characters. In that case, the character constant will be converted to a `char` whose value is the same as the ASCII value of the character.

That was a mouthful. Let's just have an example:

```
#include <stdio.h>

int main(void)
{
    char a = 10;
    char b = 'B';  // ASCII value 66

    printf("%d\n", a + b);  // 76!
}
```

This depends on your execution environment and the character set used[6]. One of the most popular character sets today is Unicode[7] (which is a superset of ASCII), so for your basic 0-9, A-Z, a-z and punctuation, you'll almost certainly get the ASCII values out of them.

14.3 More Integer Types: `short`, `long`, `long long`

So far we've just generally been using two integer types:

- `char`
- `int`

and we recently learned about the unsigned variants of the integer types. And we learned that `char` was secretly a small `int` in disguise. So we know the `int`s can come in multiple bit sizes.

[5] https://en.wikipedia.org/wiki/ASCII
[6] https://en.wikipedia.org/wiki/List_of_information_system_character_sets
[7] https://en.wikipedia.org/wiki/Unicode

But there are a couple more integer types we should look at, and the *minimum* minimum and maximum values they can hold.

Yes, I said ``minimum" twice. The spec says that these types will hold numbers of *at least* these sizes, so your implementation might be different. The header file `<limits.h>` defines macros that hold the minimum and maximum integer values; rely on that to be sure, and *never hardcode or assume these values*.

These additional types are `short int`, `long int`, and `long long int`. Commonly, when using these types, C developers leave the `int` part off (e.g. `long long`), and the compiler is perfectly happy.

```
// These two lines are equivalent:
long long int x;
long long x;

// And so are these:
short int x;
short x;
```

Let's take a look at the integer data types and sizes in ascending order, grouped by signedness.

Type	Minimum Bytes	Minimum Value	Maximum Value
`char`	1	-127 or 0	127 or 255[8]
`signed char`	1	-127	127
`short`	2	-32767	32767
`int`	2	-32767	32767
`long`	4	-2147483647	2147483647
`long long`	8	-9223372036854775807	9223372036854775807
`unsigned char`	1	0	255
`unsigned short`	2	0	65535
`unsigned int`	2	0	65535
`unsigned long`	4	0	4294967295
`unsigned long long`	8	0	18446744073709551615

There is no `long long long` type. You can't just keep adding `long`s like that. Don't be silly.

> Two's complement fans might have noticed something funny about those numbers. Why does, for example, the `signed char` stop at -127 instead of -128? Remember: these are only the minimums required by the spec. Some number representations (like sign and magnitude[9]) top off at ±127.

Let's run the same table on my 64-bit, two's complement system and see what comes out:

Type	My Bytes	Minimum Value	Maximum Value
`char`	1	-128	127[10]
`signed char`	1	-128	127
`short`	2	-32768	32767
`int`	4	-2147483648	2147483647
`long`	8	-9223372036854775808	9223372036854775807
`long long`	8	-9223372036854775808	9223372036854775807
`unsigned char`	1	0	255
`unsigned short`	2	0	65535
`unsigned int`	4	0	4294967295

[8] Depends on if a `char` defaults to `signed char` or `unsigned char`

[9] https://en.wikipedia.org/wiki/Signed_number_representations#Signed_magnitude_representation

Type	My Bytes	Minimum Value	Maximum Value
`unsigned long`	8	0	18446744073709551615
`unsigned long long`	8	0	18446744073709551615

That's a little more sensible, but we can see how my system has larger limits than the minimums in the specification.

So what are the macros in `<limits.h>`?

Type	Min Macro	Max Macro
`char`	`CHAR_MIN`	`CHAR_MAX`
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`
`short`	`SHRT_MIN`	`SHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`
`long long`	`LLONG_MIN`	`LLONG_MAX`
`unsigned char`	`0`	`UCHAR_MAX`
`unsigned short`	`0`	`USHRT_MAX`
`unsigned int`	`0`	`UINT_MAX`
`unsigned long`	`0`	`ULONG_MAX`
`unsigned long long`	`0`	`ULLONG_MAX`

Notice there's a way hidden in there to determine if a system uses signed or unsigned `char`s. If `CHAR_MAX == UCHAR_MAX`, it must be unsigned.

Also notice there's no minimum macro for the `unsigned` variants---they're just `0`.

14.4 More Float: `double` and `long double`

Let's see what the spec has to say about floating point numbers in §5.2.4.2.2¶1-2:

> The following parameters are used to define the model for each floating-point type:

Parameter	Definition
s	sign (± 1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum e_{min} and a maximum e_{max})
p	precision (the number of base-b digits in the significand)
f_k	nonnegative integers less than b (the significand digits)

> A *floating-point number* (x) is defined by the following model:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \le e \le e_{max}$$

[10] My `char` is signed.

I hope that cleared it right up for you.

Okay, fine. Let's step back a bit and see what's practical.

Note: we refer to a bunch of macros in this section. They can be found in the header `<float.h>`.

Floating point number are encoded in a specific sequence of bits (IEEE-754 format[11] is tremendously popular) in bytes.

Diving in a bit more, the number is basically represented as the *significand* (which is the number part---the significant digits themselves, also sometimes referred to as the *mantissa*) and the *exponent*, which is what power to raise the digits to. Recall that a negative exponent can make a number smaller.

Imagine we're using 10 as a number to raise by an exponent. We could represent the following numbers by using a significand of 12345, and exponents of -3, 4, and 0 to encode the following floating point values:

$12345 \times 10^{-3} = 12.345$

$12345 \times 10^{4} = 123450000$

$12345 \times 10^{0} = 12345$

For all those numbers, the significand stays the same. The only difference is the exponent.

On your machine, the base for the exponent is probably 2, not 10, since computers like binary. You can check it by printing the `FLT_RADIX` macro.

So we have a number that's represented by a number of bytes, encoded in some way. Because there are a limited number of bit patterns, a limited number of floating point numbers can be represented.

But more particularly, only a certain number of significant decimal digits can be represented accurately.

How can you get more? You can use larger data types!

And we have a couple of them. We know about `float` already, but for more precision we have `double`. And for even more precision, we have `long double` (unrelated to `long int` except by name).

The spec doesn't go into how many bytes of storage each type should take, but on my system, we can see the relative size increases:

Type	`sizeof`
`float`	4
`double`	8
`long double`	16

So each of the types (on my system) uses those additional bits for more precision.

But *how much* precision are we talking, here? How many decimal numbers can be represented by these values?

Well, C provides us with a bunch of macros in `<float.h>` to help us figure that out.

It gets a little wonky if you are using a base-2 (binary) system for storing the numbers (which is virtually everyone on the planet, probably including you), but bear with me while we figure it out.

14.4.1 How Many Decimal Digits?

The million dollar question is, ``How many significant decimal digits can I store in a given floating point type so that I get out the same decimal number when I print it?"

[11] https://en.wikipedia.org/wiki/IEEE_754

The number of decimal digits you can store in a floating point type and surely get the same number back out when you print it is given by these macros:

Type	Decimal Digits You Can Store	Minimum
`float`	`FLT_DIG`	6
`double`	`DBL_DIG`	10
`long double`	`LDBL_DIG`	10

On my system, `FLT_DIG` is 6, so I can be sure that if I print out a 6 digit `float`, I'll get the same thing back. (It could be more digits---some numbers will come back correctly with more digits. But 6 is definitely coming back.)

For example, printing out `floats` following this pattern of increasing digits, we apparently make it to 8 digits before something goes wrong, but after that we're back to 7 correct digits.

```
0.12345
0.123456
0.1234567
0.12345678
0.123456791  <-- Things start going wrong
0.1234567910
```

Let's do another demo. In this code we'll have two `floats` that both hold numbers that have `FLT_DIG` significant decimal digits[12]. Then we add those together, for what should be 12 significant decimal digits. But that's more than we can store in a `float` and correctly recover as a string---so we see when we print it out, things start going wrong after the 7th significant digit.

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

int main(void)
{
    // Both these numbers have 6 significant digits, so they can be
    // stored accurately in a float:

    float f = 3.14159f;
    float g = 0.00000265358f;

    printf("%.5f\n", f);   // 3.14159       -- correct!
    printf("%.11f\n", g);  // 0.00000265358 -- correct!

    // Now add them up
    f += g;                // 3.14159265358 is what f _should_ be

    printf("%.11f\n", f);  // 3.14159274101 -- wrong!
}
```

(The above code has an `f` after the numeric constants---this indicates that the constant is type `float`, as opposed to the default of `double`. More on this later.)

Remember that `FLT_DIG` is the safe number of digits you can store in a `float` and retrieve correctly.

Sometimes you might get one or two more out of it. But sometimes you'll only get `FLT_DIG` digits back. The

[12]This program runs as its comments indicate on a system with `FLT_DIG` of `6` that uses IEEE-754 base-2 floating point numbers. Otherwise, you might get different output.

sure thing: if you store any number of digits up to and including `FLT_DIG` in a `float`, you're sure to get them back correctly.

So that's the story. `FLT_DIG`. The End.

…Or is it?

14.4.2 Converting to Decimal and Back

But storing a base 10 number in a floating point number and getting it back out is only half the story.

Turns out floating point numbers can encode numbers that require more decimal places to print out completely. It's just that your big decimal number might not map to one of those numbers.

That is, when you look at floating point numbers from one to the next, there's a gap. If you try to encode a decimal number in that gap, it'll use the closest floating point number. That's why you can only encode `FLT_DIG` for a `float`.

But what about those floating point numbers that *aren't* in the gap? How many places do you need to print those out accurately?

Another way to phrase this question is for any given floating point number, how many decimal digits do I have to preserve if I want to convert the decimal number back into an identical floating point number? That is, how many digits do I have to print in base 10 to recover **all** the digits in base 2 in the original number?

Sometimes it might only be a few. But to be sure, you'll want to convert to decimal with a certain safe number of decimal places. That number is encoded in the following macros:

Macro	Description
`FLT_DECIMAL_DIG`	Number of decimal digits encoded in a `float`.
`DBL_DECIMAL_DIG`	Number of decimal digits encoded in a `double`.
`LDBL_DECIMAL_DIG`	Number of decimal digits encoded in a `long double`.
`DECIMAL_DIG`	Same as the widest encoding, `LDBL_DECIMAL_DIG`.

Let's see an example where `DBL_DIG` is 15 (so that's all we can have in a constant), but `DBL_DECIMAL_DIG` is 17 (so we have to convert to 17 decimal numbers to preserve all the bits of the original `double`).

Let's assign the 15 significant digit number `0.123456789012345` to `x`, and let's assign the 1 significant digit number `0.0000000000000006` to `y`.

```
x is exact: 0.12345678901234500    Printed to 17 decimal places
y is exact: 0.00000000000000060
```

But let's add them together. This should give `0.1234567890123456`, but that's more than `DBL_DIG`, so strange things might happen… let's look:

```
x + y not quite right: 0.12345678901234559    Should end in 4560!
```

That's what we get for printing more than `DBL_DIG`, right? But check this out… that number, above, is exactly representable as it is!

If we assign `0.12345678901234559` (17 digits) to `z` and print it, we get:

```
z is exact: 0.12345678901234559   17 digits correct! More than DBL_DIG!
```

If we'd truncated `z` down to 15 digits, it wouldn't have been the same number. That's why to preserve all the bits of a `double`, we need `DBL_DECIMAL_DIG` and not just the lesser `DBL_DIG`.

All that being said, it's clear that when we're messing with decimal numbers in general, it's not safe to print more than `FLT_DIG`, `DBL_DIG`, or `LDBL_DIG` digits to be sensible in relation to the original base 10 numbers and any subsequent math.

But when converting from `float` to a decimal representation and *back* to `float`, definitely use `FLT_DECIMAL_DIG` to do that so that all the bits are preserved exactly.

14.5 Constant Numeric Types

When you write down a constant number, like `1234`, it has a type. But what type is it? Let's look at how C decides what type the constant is, and how to force it to choose a specific type.

14.5.1 Hexadecimal and Octal

In addition to good ol' decimal like Grandma used to bake, C also supports constants of different bases.

If you lead a number with `0x`, it is read as a hex number:

```
int a = 0x1A2B;   // Hexadecimal
int b = 0x1a2b;   // Case doesn't matter for hex digits

printf("%x", a);  // Print a hex number, "1a2b"
```

If you lead a number with a `0`, it is read as an octal number:

```
int a = 012;

printf("%o\n", a);  // Print an octal number, "12"
```

This is particularly problematic for beginner programmers who try to pad decimal numbers on the left with `0` to line things up nice and pretty, inadvertently changing the base of the number:

```
int x = 11111;  // Decimal 11111
int y = 00111;  // Decimal 73 (Octal 111)
int z = 01111;  // Decimal 585 (Octal 1111)
```

14.5.1.1 A Note on Binary

An unofficial extension[13] in many C compilers allows you to represent a binary number with a `0b` prefix:

```
int x = 0b101010;    // Binary 101010

printf("%d\n", x);   // Prints 42 decimal
```

There's no `printf()` format specifier for printing a binary number. You have to do it a character at a time with bitwise operators.

14.5.2 Integer Constants

You can force a constant integer to be a certain type by appending a suffix to it that indicates the type.

We'll do some assignments to demo, but most often devs leave off the suffixes unless needed to be precise. The compiler is pretty good at making sure the types are compatible.

[13] It's really surprising to me that C doesn't have this in the spec yet. In the C99 Rationale document, they write, ``A proposal to add binary constants was rejected due to lack of precedent and insufficient utility.'' Which seems kind of silly in light of some of the other features they kitchen-sinked in there! I'll bet one of the next releases has it.

```
int           x = 1234;
long int      x = 1234L;
long long int x = 1234LL

unsigned int           x = 1234U;
unsigned long int      x = 1234UL;
unsigned long long int x = 1234ULL;
```

The suffix can be uppercase or lowercase. And the `U` and `L` or `LL` can appear either one first.

Type	Suffix
`int`	None
`long int`	`L`
`long long int`	`LL`
`unsigned int`	`U`
`unsigned long int`	`UL`
`unsigned long long int`	`ULL`

I mentioned in the table that ``no suffix" means `int`... but it's actually more complex than that.

So what happens when you have an unsuffixed number like:

```
int x = 1234;
```

What type is it?

What C will generally do is choose the smallest type from `int` up that can hold the value.

But specifically, that depends on the number's base (decimal, hex, or octal), as well.

The spec has a great table indicating which type gets used for what unsuffixed value. In fact, I'm just going to copy it wholesale right here.

C11 §6.4.4.1¶5 reads, ``The type of an integer constant is the first of the first of the corresponding list in which its value can be represented."

And then goes on to show this table:

Suffix	Decimal Constant	Octal or Hexadecimal Constant

none	`int`
	`long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

Suffix	Decimal Constant	Octal or Hexadecimal Constant

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

What that's saying is that, for example, if you specify a number like `123456789U`, first C will see if it can be `unsigned int`. If it doesn't fit there, it'll try `unsigned long int`. And then `unsigned long long int`. It'll use the smallest type that can hold the number.

14.5.3 Floating Point Constants

You'd think that a floating point constant like `1.23` would have a default type of `float`, right?

Surprise! Turns out unsuffiexed floating point numbers are type `double`! Happy belated birthday!

You can force it to be of type `float` by appending an `f` (or `F`---it's case-insensitive). You can force it to be of type `long double` by appending `l` (or `L`).

Type	Suffix
`float`	`F`
`double`	None
`long double`	`L`

For example:

```
float x       = 3.14f;
double x      = 3.14;
long double x = 3.14L;
```

This whole time, though, we've just been doing this, right?

```
float x = 3.14;
```

Isn't the left a `float` and the right a `double`? Yes! But C's pretty good with automatic numeric conversions, so it's more common to have an unsuffixed floating point constant than not. More on that later.

14.5.3.1 Scientific Notation

Remember earlier when we talked about how a floating point number can be represented by a significand, base, and exponent?

Well, there's a common way of writing such a number, shown here followed by it's more recognizable equivalent which is what you get when you actually run the math:

$1.2345 \times 10^3 = 1234.5$

Writing numbers in the form $s \times b^e$ is called *scientific notation*[14]. In C, these are written using ``E notation", so these are equivalent:

Scientific Notation	E notation
$1.2345 \times 10^{-3} = 0.0012345$	`1.2345e-3`
$1.2345 \times 10^8 = 123450000$	`1.2345e+8`

You can print a number in this notation with `%e`:

```
printf("%e\n", 123456.0);  // Prints 1.234560e+05
```

A couple little fun facts about scientific notation:

- You don't have to write them with a single leading digit before the decimal point. Any number of numbers can go in front.

  ```
  double x = 123.456e+3;  // 123456
  ```

  However, when you print it, it will change the exponent so there is only one digit in front of the decimal point.

- The plus can be left off the exponent, as it's default, but this is uncommon in practice from what I've seen.

  ```
  1.2345e10 == 1.2345e+10
  ```

- You can apply the `F` or `L` suffixes to E-notation constants:

  ```
  1.2345e10F
  1.2345e10L
  ```

14.5.3.2 Hexadecimal Floating Point Constants

But wait, there's more floating to be done!

Turns out there are hexadecimal floating point constants, as well!

These work similar to decimal floating point numbers, but they begin with a `0x` just like integer numbers.

The catch is that you *must* specify an exponent, and this exponent produces a power of 2. That is: 2^x.

And then you use a `p` instead of an `e` when writing the number:

So `0xa.1p3` is $10.0625 \times 2^3 == 80.5$.

When using floating point hex constants, We can print hex scientific notation with `%a`:

```
double x = 0xa.1p3;

printf("%a\n", x);  // 0x1.42p+6
printf("%f\n", x);  // 80.500000
```

[14] https://en.wikipedia.org/wiki/Scientific_notation

Chapter 15

Types III: Conversions

In this chapter, we want to talk all about converting from one type to another. C has a variety of ways of doing this, and some might be a little different that you're used to in other languages.

Before we talk about how to make conversions happen, let's talk about how they work when they *do* happen.

15.1 String Conversions

Unlike many languages, C doesn't do string-to-number (and vice-versa) conversions in quite as streamlined a manner as it does numeric conversions.

For these, we'll have to call functions to do the dirty work.

15.1.1 Numeric Value to String

When we want to convert a number to a string, we can use either `sprintf()` (pronounced *SPRINT-f*) or `snprintf()` (*s-n-print-f*)[1]

These basically work like `printf()`, except they output to a string instead, and you can print that string later, or whatever.

For example, turning part of the value π into a string:

```
#include <stdio.h>

int main(void)
{
    char s[10];
    float f = 3.14159;

    // Convert "f" to string, storing in "s", writing at most 10 characters
    // including the NUL terminator

    snprintf(s, 10, "%f", f);

    printf("String value: %s\n", s);  // String value: 3.141590
}
```

So you can use `%d` or `%u` like you're used to for integers.

[1] They're the same except `snprintf()` allows you to specify a maximum number of bytes to output, preventing the overrunning of the end of your string. So it's safer.

15.1.2 String to Numeric Value

There are a couple families of functions to do this in C. We'll call these the `atoi` (pronounced *a-to-i*) family and the `strtol` (*stir-to-long*) family.

For basic conversion from a string to a number, try the `atoi` functions from `<stdlib.h>`. These have bad error-handling characteristics (including undefined behavior if you pass in a bad string), so use them carefully.

Function	Description
`atoi`	String to `int`
`atof`	String to `float`
`atol`	String to `long int`
`atoll`	String to `long long int`

Though the spec doesn't cop to it, the `a` at the beginning of the function stands for ASCII[2], so really `atoi()` is ``ASCII-to-integer", but saying so today is a bit ASCII-centric.

Here's an example converting a string to a `float`:

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

int main(void)
{
    char *pi = "3.14159";
    float f;

    f = atof(pi);

    printf("%f\n", f);
}
```

But, like I said, we get undefined behavior from weird things like this:

```
int x = atoi("what");  // "What" ain't no number I ever heard of
```

(When I run that, I get `0` back, but you really shouldn't count on that in any way. You could get something completely different.)

For better error handling characteristics, let's check out all those `strtol` functions, also in `<stdlib.h>`. Not only that, but they convert to more types and more bases, too!

Function	Description
`strtol`	String to `long int`
`strtoll`	String to `long long int`
`strtoul`	String to `unsigned long int`
`strtoull`	String to `unsigned long long int`
`strtof`	String to `float`
`strtod`	String to `double`
`strtold`	String to `long double`

These functions all follow a similar pattern of use, and are a lot of people's first experience with pointers to pointers! But never fret---it's easier than it looks.

[2] https://en.wikipedia.org/wiki/ASCII

Let's do an example where we convert a string to an `unsigned long`, discarding error information (i.e. information about bad characters in the input string):

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

int main(void)
{
    char *s = "3490";

    // Convert string s, a number in base 10, to an unsigned long int.
    // NULL means we don't care to learn about any error information.

    unsigned long int x = strtoul(s, NULL, 10);

    printf("%lu\n", x);  // 3490
}
```

Notice a couple things there. Even though we didn't deign to capture any information about error characters in the string, `strtoul()` won't give us undefined behavior; it will just return `0`.

Also, we specified that this was a decimal (base 10) number.

Does this mean we can convert numbers of different bases? Sure! Let's do binary!

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

int main(void)
{
    char *s = "101010";  // What's the meaning of this number?

    // Convert string s, a number in base 2, to an unsigned long int.

    unsigned long int x = strtoul(s, NULL, 2);

    printf("%lu\n", x);  // 42
}
```

OK, that's all fun and games, but what's with that `NULL` in there? What's that for?

That helps us figure out if an error occurred in the processing of the string. It's a pointer to a pointer to a `char`, which sounds scary, but isn't once you wrap your head around it.

Let's do an example where we feed in a deliberately bad number, and we'll see how `strtol()` lets us know where the first invalid digit is.

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

int main(void)
{
    char *s = "34x90";  // "x" is not a valid digit in base 10!
    char *badchar;

    // Convert string s, a number in base 10, to an unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);
```

```
    // It tries to convert as much as possible, so gets this far:

    printf("%lu\n", x);  // 34

    // But we can see the offending bad character because badchar
    // points to it!

    printf("Invalid character: %c\n", *badchar);  // "x"
}
```

So there we have `strtoul()` modifying what `badchar` points to in order to show us where things went wrong[3].

But what if nothing goes wrong? In that case, `badchar` will point to the `NUL` terminator at the end of the string. So we can test for it:

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

int main(void)
{
    char *s = "3490";  // "x" is not a valid digit in base 10!
    char *badchar;

    // Convert string s, a number in base 10, to an unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);

    // Check if things went well

    if (*badchar == '\0') {
        printf("Success! %lu\n", x);
    } else  {
        printf("Partial conversion: %lu\n", x);
        printf("Invalid character: %c\n", *badchar);
    }
}
```

So there you have it. The `atoi()`-style functions are good in a controlled pinch, but the `strtol()`-style functions give you far more control over error handling and the base of the input.

15.2 `char` Conversions

What if you have a single character with a digit in it, like `'5'`… Is that the same as the value `5`?

Let's try it and see.

```
printf("%d %d\n", 5, '5');
```

On my UTF-8 system, this prints:

```
5 53
```

[3]We have to pass a pointer to `badchar` to `strtoul()` or it won't be able to modify it in any way we can see, analogous to why you have to pass a pointer to an `int` to a function if you want that function to be able to change that value of that `int`.

So… no. And 53? What is that? That's the UTF-8 (and ASCII) code point for the character symbol `'5'`[4]

So how do we convert the character `'5'` (which apparently has value 53) into the value `5`?

With one clever trick, that's how!

The C Standard guarantees that these character will have code points that are in sequence and in this order:

```
0  1  2  3  4  5  6  7  8  9
```

Ponder for a second--how can we use that? Spoilers ahead…

Let's take a look at the characters and their code points in UTF-8:

```
0  1  2  3  4  5  6  7  8  9
48 49 50 51 52 53 54 55 56 57
```

You see there that `'5'` is `53`, just like we were getting. And `'0'` is `48`.

So we can subtract `'0'` from any digit character to get its numeric value:

```
char c = '6';

int x = c;  // x has value 54, the code point for '6'

int y = c - '0'; // y has value 6, just like we want
```

And we can convert the other way, too, just by adding the value on.

```
int x = 6;

char c = x + '0';  // c has value 54

printf("%d\n", c);  // prints 54
printf("%c\n", c);  // prints 6 with %c
```

You might think this is a weird way to do this conversion, and by today's standards, it certainly is. But back in the olden days when computers were made literally out of wood, this was the method for doing this conversion. And it wasn't broke, so C never fixed it.

15.3 Numeric Conversions

15.3.1 Boolean

If you convert a zero to `bool`, the result is `0`. Otherwise it's `1`.

15.3.2 Integer to Integer Conversions

If an integer type is converted to unsigned and doesn't fit in it, the unsigned result wraps around odometer-style until it fits in the unsigned[5].

If an integer type is converted to a signed number and doesn't fit, the result is implementation-defined! Something documented will happen, but you'll have to look it up[6]

[4] Each character has a value associated with it for any given character encoding scheme.

[5] In practice, what's probably happening on your implementation is that the high-order bits are just being dropped from the result, so a 16-bit number `0x1234` being converted to an 8-bit number ends up as `0x0034`, or just `0x34`.

[6] Again, in practice, what will likely happen on your system is that the bit pattern for the original will be truncated and then just used to represent the signed number, two's complement. For example, my system takes an `unsigned char` of `192` and converts it to `signed char -64`. In two's complement, the bit pattern for both these numbers is binary `11000000`.

15.3.3 Integer and Floating Point Conversions

If a floating point type is converted to an integer type, the fractional part is discarded with prejudice[7].

But---and here's the catch---if the number is too large to fit in the integer, you get undefined behavior. So don't do that.

Going From integer or floating point to floating point, C makes a best effort to find the closest floating point number to the integer that it can.

Again, though, if the original value can't be represented, it's undefined behavior.

15.4 Implicit Conversions

These are conversions the compiler does automatically for you when you mix and match types.

15.4.1 The Integer Promotions

In a number of places, if an `int` can be used to represent a value from `char` or `short` (signed or unsigned), that value is *promoted* up to `int`. If it doesn't fit in an `int`, it's promoted to `unsigned int`.

This is how we can do something like this:

```
char x = 10, y = 20;
int i = x + y;
```

In that case, `x` and `y` get promoted to `int` by C before the math takes place.

The integer promotions take place during The Usual Arithmetic Conversions, with variadic functions[8], unary `+` and `-` operators, or when passing values to functions without prototypes[9].

15.4.2 The Usual Arithmetic Conversions

These are automatic conversions that C does around numeric operations that you ask for. (That's actually what they're called, by the way, by C11 §6.3.1.8.) Note that for this section, we're just talking about numeric types---strings will come later.

These conversions answer questions about what happens when you mix types, like this:

```
int x = 3 + 1.2;   // Mixing int and double
                   // 4.2 is converted to int
                   // 4 is stored in x

float y = 12 * 2;  // Mixing float and int
                   // 24 is converted to float
                   // 24.0 is stored in y
```

Do they become `int`s? Do they become `float`s? How does it work?

Here are the steps, paraphrased for easy consumption.

1. If one thing in the expression is a floating type, convert the other things to that floating type.
2. Otherwise, if both types are integer types, perform the integer promotions on each, then make the operand types as big as they need to be hold the common largest value. Sometimes this involves changing signed to unsigned.

[7] Not really---it's just discarded regularly.

[8] Functions with a variable number of arguments.

[9] This is rarely done because the compiler will complain and having a prototype is the *Right Thing* to do. I think this still works for historic reasons, before prototypes were a thing.

If you want to know the gritty details, check out C11 §6.3.1.8. But you probably don't.

Just generally remember that int types become float types if there's a floating point type anywhere in there, and the compiler makes an effort to make sure mixed integer types don't overflow.

Finally, if you convert from one floating point type to another, the compiler will try to make an exact conversion. If it can't, it'll do the best approximation it can. If the number is too large to fit in the type you're converting into, *boom*: undefined behavior!

15.4.3 `void*`

The `void*` type is interesting because it can be converted from or to any pointer type.

```
int x = 10;

void *p = &x;  // &x is type int*, but we store it in a void*

int *q = p;    // p is void*, but we store it in an int*
```

15.5 Explicit Conversions

These are conversions from type to type that you have to ask for; the compiler won't do it for you.

You can convert from one type to another by assigning one type to another with an `=`.

You can also convert explicitly with a *cast*.

15.5.1 Casting

You can explicitly change the type of an expression by putting a new type in parentheses in front of it. Some C devs frown on the practice unless absolutely necessary, but it's likely you'll come across some C code with these in it.

Let's do an example where we want to convert an `int` into a `long` so that we can store it in a `long`.

Note: this example is contrived and the cast in this case is completely unnecessary because the `x + 12` expression would automatically be changed to `long int` to match the wider type of `y`.

```
int x = 10;
long int y = (long int)x + 12;
```

In that example, even those `x` was type `int` before, the expression `(long int)x` has type `long int`. We say, ``We cast `x` to `long int`."

More commonly, you might see a cast being used to convert a `void*` into a specific pointer type so it can be dereferenced.

A callback from the built-in `qsort()` function might display this behavior since it has `void*`s passed into it:

```
int compar(const void *elem1, const void *elem2)
{
    if (*((const int*)elem2) > *((const int*)elem1)) return 1;
    if (*((const int*)elem2) < *((const int*)elem1)) return -1;
    return 0;
}
```

But you could also clearly write it with an assignment:

```
int compar(const void *elem1, const void *elem2)
{
    const int *e1 = elem1;
    const int *e2 = elem2;

    return *e2 - *e1;
}
```

One place you'll see casts more commonly is to avoid a warning when printing pointer values with the rarely-used `%p` which gets picky with anything other than a `void*`:

```
int x = 3490;
int *p = &x;

printf("%p\n", p);
```

generates this warning:

```
warning: format ‘%p’ expects argument of type ‘void *’, but argument
         2 has type ‘int *’
```

You can fix it with a cast:

```
printf("%p\n", (void *)p);
```

Another place is with explicit pointer changes, if you don't want to use an intervening `void*`, but these are also pretty uncommon:

```
long x = 3490;
long *p = &x;
unsigned char *c = (unsigned char *)p;
```

A third place it's often required is with the character conversion functions in `<ctype.h>`[10] where you should cast questionably-signed values to `unsigned char` to avoid undefined behavior.

Again, casting is rarely *needed* in practice. If you find yourself casting, there might be another way to do the same thing, or maybe you're casting unnecessarily.

Or maybe it is necessary. Personally, I try to avoid it, but am not afraid to use it if I have to.

[10] https://beej.us/guide/bgclr/html/split/ctype.html

Chapter 16

Types IV: Qualifiers and Specifiers

Now that we have some more types under our belts, turns out we can give these types some additional attributes that control their behavior. These are the *type qualifiers* and *storage-class specifiers*.

16.1 Type Qualifiers

These are going to allow you to declare constant values, and also to give the compiler optimization hints that it can use.

16.1.1 `const`

This is the most common type qualifier you'll see. It means the variable is constant, and any attempt to modify it will result in a very angry compiler.

```
const int x = 2;

x = 4;  // COMPILER PUKING SOUNDS, can't assign to a constant
```

You can't change a `const` value.

Often you see `const` in parameter lists for functions:

```
void foo(const int x)
{
    printf("%d\n", x + 30);  // OK, doesn't modify "x"
}
```

16.1.1.1 `const` and Pointers

This one gets a little funky, because there are two usages that have two meanings when it comes to pointers.

For one thing, we can make it so you can't change the thing the pointer points to. You do this by putting the `const` up front with the type name (before the asterisk) in the type declaration.

```
int x[] = {10, 20};
const int *p = x;

p++;  // We can modify p, no problem

*p = 30; // Compiler error! Can't change what it points to
```

Somewhat confusingly, these two things are equivalent:

```
const int *p;  // Can't modify what p points to
int const *p;  // Can't modify what p points to, just like the previous line
```

Great, so we can't change the thing the pointer points to, but we can change the pointer itself. What if we want the other way around? We want to be able to change what the pointer points to, but *not* the pointer itself?

Just move the `const` after the asterisk in the declaration:

```
int *const p;   // We can't modify "p" with pointer arithmetic

p++;  // Compiler error!
```

But we can modify what they point to:

```
int x = 10;
int *const p = &x;

*p = 20;   // Set "x" to 20, no problem
```

You can also do make both things `const`:

```
const int *const p;  // Can't modify p or *p!
```

Finally, if you have multiple levels of indirection, you should `const` the appropriate levels. Just because a pointer is `const`, doesn't mean the pointer it points to must also be. You can explicitly set them like in the following examples:

```
char **p;
p++;     // OK!
(*p)++;  // OK!

char **const p;
p++;     // Error!
(*p)++;  // OK!

char *const *p;
p++;     // OK!
(*p)++;  // Error!

char *const *const p;
p++;     // Error!
(*p)++;  // Error!
```

16.1.1.2 `const` Correctness

One more thing I have to mention is that the compiler will warn on something like this:

```
const int x = 20;
int *p = &x;
```

saying something to the effect of:

```
initialization discards 'const' qualifier from pointer type target
```

What's happening there?

Well, we need to look at the types on either side of the assignment:

```
    const int x = 20;
    int *p = &x;
//    ^       ^
//    |       |
//  int*    const int*
```

The compiler is warning us that the value on the right side of the assignment is `const`, but the one of the left is not. And the compiler is letting us know that it is discarding the ``const-ness'' of the expression on the right.

That is, we *can* still try to do the following, but it's just wrong. The compiler will warn, and it's undefined behavior:

```
const int x = 20;
int *p = &x;

*p = 40;  // Undefined behavior--maybe it modifies "x", maybe not!

printf("%d\n", x);  // 40, if you're lucky
```

16.1.2 `restrict`

TLDR: you never have to use this and you can ignore it every time you see it. If you use it correctly, you will likely realize some performance gain. If you use it incorrectly, you will realize undefined behavior.

`restrict` is a hint to the compiler that a particular piece of memory will only be accessed by one pointer and never another. (That is, there will be no aliasing of the particular object the `restrict` pointer points to.) If a developer declares a pointer to be `restrict` and then accesses the object it points to in another way (e.g. via another pointer), the behavior is undefined.

Basically you're telling C, ``Hey---I guarantee that this one single pointer is the only way I access this memory, and if I'm lying, you can pull undefined behavior on me.''

And C uses that information to perform certain optimizations. For instance, if you're dereferencing the `restrict` pointer repeatedly in a loop, C might decide to cache the result in a register and only store the final result once the loop completes. If any other pointer referred to that same memory and accessed it in the loop, the results would not be accurate.

(Note that `restrict` has no effect if the object pointed to is never written to. It's all about optimizations surrounding writes to memory.)

Let's write a function to swap two variables, and we'll use the `restrict` keyword to assure C that we'll never pass in pointers to the same thing. And then let's blow it and try passing in pointers to the same thing.

```
1  void swap(int *restrict a, int *restrict b)
2  {
3      int t;
4
5      t = *a;
6      *a = *b;
7      *b = t;
8  }
9
10 int main(void)
11 {
12     int x = 10, y = 20;
13
14     swap(&x, &y);  // OK! "a" and "b", above, point to different things
```

```
15
16      swap(&x, &x);  // Undefined behavior! "a" and "b" point to the same thing
17  }
```

If we were to take out the `restrict` keywords, above, that would allow both calls to work safely. But then the compiler might not be able to optimize.

`restrict` has block scope, that is, the restriction only lasts for the scope it's used. If it's in a parameter list for a function, it's in the block scope of that function.

If the restricted pointer points to an array, it only applies to the individual objects in the array. Other pointers could read and write from the array as long as they didn't read or write any of the same elements as the restricted one.

If it's outside any function in file scope, the restriction covers the entire program.

You're likely to see this in library functions like `printf()`:

```
int printf(const char * restrict format, ...);
```

Again, that's just telling the compiler that inside the `printf()` function, there will be only one pointer that refers to any part of that `format` string.

One last note: if you're using array notation in your function parameter for some reason instead of pointer notation, you can use `restrict` like so:

```
void foo(int p[restrict])     // With no size

void foo(int p[restrict 10])  // Or with a size
```

But pointer notation would be more common.

16.1.3 `volatile`

You're unlikely to see or need this unless you're dealing with hardware directly.

`volatile` tells the compiler that a value might change behind its back and should be looked up every time.

An example might be where the compiler is looking in memory at an address that continuously updates behind the scenes, e.g. some kind of hardware timer.

If the compiler decides to optimize that and store the value in a register for a protracted time, the value in memory will update and won't be reflected in the register.

By declaring something `volatile`, you're telling the compiler, ``Hey, the thing this points at might change at any time for reasons outside this program code."

```
volatile int *p;
```

16.1.4 `_Atomic`

This is an optional C feature that we'll talk about in the Atomics chapter.

16.2 Storage-Class Specifiers

Storage-class specifiers are similar to type quantifiers. They give the compiler more information about the type of a variable.

16.2.1 `auto`

You barely ever see this keyword, since `auto` is the default for block scope variables. It's implied.

These are the same:

```
{
    int a;          // auto is the default...
    auto int a;     // So this is redundant
}
```

The `auto` keyword indicates that this object has *automatic storage duration*. That is, it exists in the scope in which it is defined, and is automatically deallocated when the scope is exited.

One gotcha about automatic variables is that their value is indeterminate until you explicitly initialize them. We say they're full of ``random" or ``garbage" data, though neither of those really makes me happy. In any case, you won't know what's in it unless you initialize it.

Always initialize all automatic variables before use!

16.2.2 `static`

This keyword has two meanings, depending on if the variable is file scope or block scope.

Let's start with block scope.

16.2.2.1 `static` in Block Scope

In this case, we're basically saying, ``I just want a single instance of this variable to exist, shared between calls."

That is, its value will persist between calls.

`static` in block scope with an initializer will only be initialized one time on program startup, not each time the function is called.

Let's do an example:

```
#include <stdio.h>

void counter(void)
{
    static int count = 1;  // This is initialized one time

    printf("This has been called %d time(s)\n", count);

    count++;
}

int main(void)
{
    counter();  // "This has been called 1 time(s)"
    counter();  // "This has been called 2 time(s)"
    counter();  // "This has been called 3 time(s)"
    counter();  // "This has been called 4 time(s)"
}
```

See how the value of `count` persists between calls?

One thing of note is that `static` block scope variables are initialized to `0` by default.

```
static int foo;      // Default starting value is `0`...
static int foo = 0;  // So the `0` assignment is redundant
```

Finally, be advised that if you're writing multithreaded programs, you have to be sure you don't let multiple threads trample the same variable.

16.2.2.2 `static` in File Scope

When you get out to file scope, outside any blocks, the meaning rather changes.

Variables at file scope already persist between function calls, so that behavior is already there.

Instead what `static` means in this context is that this variable isn't visible outside of this particular source file. Kinda like ``global", but only in this file.

More on that in the section about building with multiple source files.

16.2.3 `extern`

The `extern` storage-class specifier gives us a way to refer to objects in other source files.

Let's say, for example, the file `bar.c` had the following as its entirety:

```
// bar.c

int a = 37;
```

Just that. Declaring a new `int a` in file scope.

But what if we had another source file, `foo.c`, and we wanted to refer to the `a` that's in `bar.c`?

It's easy with the `extern` keyword:

```
// foo.c

extern int a;

int main(void)
{
    printf("%d\n", a);  // 37, from bar.c!

    a = 99;

    printf("%d\n", a);  // Same "a" from bar.c, but it's now 99
}
```

We could have also made the `extern int a` in block scope, and it still would have referred to the `a` in `bar.c`:

```
// foo.c

int main(void)
{
    extern int a;

    printf("%d\n", a);  // 37, from bar.c!

    a = 99;

```

```
    printf("%d\n", a);  // Same "a" from bar.c, but it's now 99
}
```

Now, if a in `bar.c` had been marked `static`. this wouldn't have worked. `static` variables at file scope are not visible outside that file.

A final note about `extern` on functions. For functions, `extern` is the default, so it's redundant. You can declare a function `static` if you only want it visible in a single source file.

16.2.4 `register`

This is a keyword to hint to the compiler that this variable is frequently-used, and should be made as fast as possible to access. The compiler is under no obligation to agree to it.

Now, modern C compiler optimizers are pretty effective at figuring this out themselves, so it's rare to see these days.

But if you must:

```
#include <stdio.h>

int main(void)
{
    register int a;   // Make "a" as fast to use as possible.

    for (a = 0; a < 10; a++)
        printf("%d\n", a);
}
```

It does come at a price, however. You can't take the address of a register:

```
register int a;
int *p = &a;    // COMPILER ERROR! Can't take address of a register
```

The same applies to any part of an array:

```
register int a[] = {11, 22, 33, 44, 55};
int *p = a;  // COMPILER ERROR! Can't take address of a[0]
```

Or dereferencing part of an array:

```
register int a[] = {11, 22, 33, 44, 55};

int a = *(a + 2);  // COMPILER ERROR! Address of a[0] taken
```

Interestingly, for the equivalent with array notation, gcc only warns:

```
register int a[] = {11, 22, 33, 44, 55};

int a = a[2];  // COMPILER WARNING!
```

with:

```
warning: ISO C forbids subscripting ‘register’ array
```

The fact that you can't take the address of a register variable frees the compiler up to make optimizations around that assumption if it hasn't figured them out already. Also adding `register` to a `const` variable prevents one from accidentally passing its pointer to another function that willfully ignore its constness[1].

[1] https://gustedt.wordpress.com/2010/08/17/a-common-misconception-the-register-keyword/

A bit of historic backstory, here: deep inside the CPU are little dedicated ``variables'' called *registers*[2]. They are super fast to access compared to RAM, so using them gets you a speed boost. But they're not in RAM, so they don't have an associated memory address (which is why you can't take the address-of or get a pointer to them).

But, like I said, modern compilers are really good at producing optimal code, using registers whenever possible regardless of whether or not you specified the `register` keyword. Not only that, but the spec allows them to just treat it as if you'd typed `auto`, if they want. So no guarantees.

16.2.5 `_Thread_local`

When you're using multiple threads and you have some variables in either global or `static` block scope, this is a way to make sure that each thread gets its own copy of the variable. This'll help you avoid race conditions and threads stepping on each other's toes.

If you're in block scope, you have to use this along with either `extern` or `static`.

Also, if you include `<threads.h>`, you can use the rather more palatable `thread_local` as an alias for the uglier `_Thread_local`.

More information can be found in the Threads section.

[2]https://en.wikipedia.org/wiki/Processor_register

Chapter 17

Multifile Projects

So far we've been looking at toy programs that for the most part fit in a single file. But complex C programs are made up of many files that are all compiled and linked together into a single executable.

In this chapter we'll check out some of the common patterns and practices for putting together larger projects.

17.1 Includes and Function Prototypes

A really common situation is that some of your functions are defined in one file, and you want to call them from another.

This actually works out of the box with a warning… let's first try it and then look at the right way to fix the warning.

For these examples, we'll put the filename as the first comment in the source.

To compile them, you'll need to specify all the sources on the command line:

```
# output file   source files
#     v            v
#   |----| |---------|
gcc -o foo foo.c bar.c
```

In that examples, `foo.c` and `bar.c` get built into the executable named `foo`.

So let's take a look at the source file `bar.c`:

```
// File bar.c

int add(int x, int y)
{
    return x + y;
}
```

And the file `foo.c` with main in it:

```
// File foo.c

#include <stdio.h>

int main(void)
{
```

```
    printf("%d\n", add(2, 3));  // 5!
}
```

See how from `main()` we call `add()`---but `add()` is in a completely different source file! It's in `bar.c`, while the call to it is in `foo.c`!

If we build this with:

```
gcc -o foo foo.c bar.c
```

we get this error:

```
error: implicit declaration of function 'add' is invalid in C99
```

(Or you might get a warning. Which you should not ignore. Never ignore warnings in C; address them all.)

If you recall from the section on prototypes, implicit declarations are banned in modern C and there's no legitimate reason to introduce them into new code. We should fix it.

What `implicit declaration` means is that we're using a function, namely `add()` in this case, without letting C know anything about it ahead of time. C wants to know what it returns, what types it takes as arguments, and things such as that.

We saw how to fix that earlier with a *function prototype*. Indeed, if we add one of those to `foo.c` before we make the call, everything works well:

```
// File foo.c

#include <stdio.h>

int add(int, int);  // Add the prototype

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

No more error!

But that's a pain---needing to type in the prototype every time you want to use a function. I mean, we used `printf()` right there and didn't need to type in a prototype; what gives?

If you remember from what back with `hello.c` at the beginning of the book, *we actually did include the prototype for `printf()`*! It's in the file `stdio.h`! And we included that with `#include`!

Can we do the same with our `add()` function? Make a prototype for it and put it in a header file?

Sure!

Header files in C have a `.h` extension by default. And they often, but not always, have the same name as their corresponding `.c` file. So let's make a `bar.h` file for our `bar.c` file, and we'll stick the prototype in it:

```
// File bar.h

int add(int, int);
```

And now let's modify `foo.c` to include that file. Assuming it's in the same directory, we include it inside double quotes (as opposed to angle brackets):

```
// File foo.c

#include <stdio.h>
```

```
#include "bar.h"  // Include from current directory

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Notice how we don't have the prototype in `foo.c` anymore---we included it from `bar.h`. Now *any* file that wants that `add()` functionality can just `#include "bar.h"` to get it, and you don't need to worry about typing in the function prototype.

As you might have guessed, `#include` literally includes the named file *right there* in your source code, just as if you'd typed it in.

And building and running:

```
./foo
5
```

Indeed, we get the result of $2 + 3$! Yay!

But don't crack open your drink of choice quite yet. We're almost there! There's just one more piece of boilerplate we have to add.

17.2 Dealing with Repeated Includes

It's not uncommon that a header file will itself `#include` other headers needed for the functionality of its corresponding C files. I mean, why not?

And it could be that you have a header `#included` multiple times from different places. Maybe that's no problem, but maybe it would cause compiler errors. And we can't control how many places `#include` it!

Even, worse we might get into a crazy situation where header `a.h` includes header `b.h`, and `b.h` includes `a.h`! It's an `#include` infinite cycle!

Trying to build such a thing gives an error:

```
error: #include nested depth 200 exceeds maximum of 200
```

What we need to do is make it so that if a file gets included once, subsequent `#includes` for that file are ignored.

The stuff that we're about to do is so common that you should just automatically do it every time you make a header file!

And the common way to do this is with a preprocessor variable that we set the first time we `#include` the file. And then for subsequent `#includes`, we first check to make sure that the variable isn't defined.

For that variable name, it's super common to take the name of the header file, like `bar.h`, make it uppercase, and replace the period with an underscore: `BAR_H`.

So put a check at the very, very top of the file where you see if it's already been included, and effectively comment the whole thing out if it has.

(Don't put a leading underscore (because a leading underscore followed by a capital letter is reserved) or a double leading underscore (because that's also reserved.))

```
#ifndef BAR_H   // If BAR_H isn't defined...
#define BAR_H   // Define it (with no particular value)

```

```
// File bar.h

int add(int, int);

#endif          // End of the #ifndef BAR_H
```

This will effectively cause the header file to be included only a single time, no matter how many places try to `#include` it.

17.3 `static` and `extern`

When it comes to multifile projects, you can make sure file-scope variables and functions are *not* visible from other source files with the `static` keyword.

And you can refer to objects in other files with `extern`.

For more info, check out the sections in the book on the `static` and `extern` storage-class specifiers.

17.4 Compiling with Object Files

This isn't part of the spec, but it's 99.999% common in the C world.

You can compile C files into an intermediate representation called *object files*. These are compiled machine code that hasn't been put into an executable yet.

Object files in Windows have a `.OBJ` extension; in Unix-likes, they're `.o`.

In gcc, we can build some like this, with the `-c` (compile only!) flag:

```
gcc -c foo.c     # produces foo.o
gcc -c bar.c     # produces bar.o
```

And then we can *link* those together into a single executable:

```
gcc -o foo foo.o bar.o
```

Voila, we've produced an executable `foo` from the two object files.

But you're thinking, why bother? Can't we just:

```
gcc -o foo foo.c bar.c
```

and kill two boids[1] with one stone?

For little programs, that's fine. I do it all the time.

But for larger programs, we can take advantage of the fact that compiling from source to object files is relatively slow, and linking together a bunch of object files is relatively fast.

This really shows with the `make` utility that only rebuilds sources that are newer than their outputs.

Let's say you had a thousand C files. You could compile them all to object files to start (slowly) and then combine all those object files into an executable (fast).

Now say you modified just one of those C source files---here's the magic: *you only have to rebuild that one object file for that source file*! And then you rebuild the executable (fast). All the other C files don't have to be touched.

In other words, by only rebuilding the object files we need to, we cut down on compilation times radically. (Unless of course you're doing a ``clean'' build, in which case all the object files have to be created.)

[1] https://en.wikipedia.org/wiki/Boids

Chapter 18

The Outside Environment

When you run a program, it's actually you talking to the shell, saying, ``Hey, please run this thing." And the shell says, ``Sure," and then tells the operating system, ``Hey, could you please make a new process and run this thing?" And if all goes well, the OS complies and your program runs.

But there's a whole world outside your program in the shell that can be interacted with from within C. We'll look at a few of those in this chapter.

18.1 Command Line Arguments

Many command line utilities accept *command line arguments*. For example, if we want to see all files that end in `.txt`, we can type something like this on a Unix-like system:

```
ls *.txt
```

(or `dir` instead of `ls` on a Windows system).

In this case, the command is `ls`, but it arguments are all all files that end with `.txt`[1].

So how can we see what is passed into program from the command line?

Say we have a program called `add` that adds all numbers passed on the command line and prints the result:

```
./add 10 30 5
45
```

That's gonna pay the bills for sure!

But seriously, this is a great tool for seeing how to get those arguments from the command line and break them down.

First, let's see how to get them at all. For this, we're going to need a new `main()`!

Here's a program that prints out all the command line arguments. For example, if we name the executable `foo`, we can run it like this:

```
./foo i like turtles
```

and we'll see this output:

[1] Historially, MS-DOS and Windows programs would do this differently than Unix. In Unix, the shell would *expand* the wildcard into all matching files before your program saw it, whereas the Microsoft variants would pass the wildcard expression into the program to deal with. In any case, there are arguments that get passed into the program.

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

It's a little weird, because the zeroth argument is the name of the executable, itself. But that's just something to get used to. The arguments themselves follow directly.

Source:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }
}
```

Whoa! What's going on with the `main()` function signature? What's `argc` and `argv`[2] (pronounced *arg-cee* and *arg-vee*)?

Let's start with the easy one first: `argc`. This is the *argument count*, including the program name, itself. If you think of all the arguments as an array of strings, which is exactly what they are, then you can think of `argc` as the length of that array, which is exactly what it is.

And so what we're doing in that loop is going through all the `argv`s and printing them out one at a time, so for a given input:

```
./foo i like turtles
```

we get a corresponding output:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

With that in mind, we should be good to go with our adder program.

Our plan:

- Look at all the command line arguments (past `argv[0]`, the program name)
- Convert them to integers
- Add them to a running total
- Print the result

Let's get to it!

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

int main(int argc, char **argv)
{
    int total = 0;

    for (int i = 1; i < argc; i++) {  // Start at 1, the first argument
```

[2] Since they're just regular parameter names, you don't actually have to call them `argc` and `argv`. But it's so very idiomatic to use those names, if you get creative, other C programmers will look at you with a suspicious eye, indeed!

```
        int value = atoi(argv[i]);    // Use strtol() for better error handling

        total += value;
    }

    printf("%d\n", total);
}
```

Sample runs:

```
$ ./add
0
$ ./add 1
1
$ ./add 1 2
3
$ ./add 1 2 3
6
$ ./add 1 2 3 4
10
```

Of course, it might puke if you pass in a non-integer, but hardening against that is left as an exercise to the reader.

18.1.1 The Last `argv` is `NULL`

One bit of fun trivia about `argv` is that after the last string is a pointer to `NULL`.

That is:

```
argv[argc] == NULL
```

is always true!

This might seem pointless, but it turns out to be useful in a couple places; we'll take a look at one of those right now.

18.1.2 The Alternate: `char **argv`

Remember that when you call a function, C doesn't differentiate between array notation and pointer notation in the function signature.

That is, these are the same:

```
void foo(char a[])
void foo(char *a)
```

Now, it's been convenient to think of `argv` as an array of strings, i.e. an array of `char*`s, so this made sense:

```
int main(int argc, char *argv[])
```

but because of the equivalence, you could also write:

```
int main(int argc, char **argv)
```

Yeah, that's a pointer to a pointer, all right! If it makes it easier, think of it as a pointer to a string. But really, it's a pointer to a value that points to a `char`.

Also recall that these are equivalent:

```
argv[i]
*(argv + i)
```

which means you can do pointer arithmetic on `argv`.

So an alternate way to consume the command line arguments might be to just walk along the `argv` array by bumping up a pointer until we hit that `NULL` at the end.

Let's modify our adder to do that:

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

int main(int argc, char **argv)
{
    int total = 0;

    // Cute trick to get the compiler to stop warning about the
    // unused variable argc:
    (void)argc;

    for (char **p = argv + 1; *p != NULL; p++) {
        int value = atoi(*p);  // Use strtol() for better error handling

        total += value;
    }

    printf("%d\n", total);
}
```

Personally, I use array notation to access `argv`, but have seen this style floating around, as well.

18.1.3 Fun Facts

Just a few more things about `argc` and `argv`.

- Some environments might not set `argv[0]` to the program name. If it's not available, `argv[0]` will be an empty string. I've never seen this happen.
- The spec is actually pretty liberal with what an implementation can do with `argv` and where those values come from. But every system I've been on works the same way, as we've discussed in this section.
- You can modify `argc`, `argv`, or any of the strings that `argv` points to. (Just don't make those strings longer than they already are!)
- On some Unix-like systems, modifying the string `argv[0]` results in the output of `ps` changing[3].

  Normally, if you have a program called `foo` that you've run with `./foo`, you might see this in the output of `ps`:

  ```
   4078 tty1     S      0:00 ./foo
  ```

  But if you modify `argv[0]` like so, being careful that the new string `"Hi!  "` is the same length as the old one `"./foo"`:

  ```
  strcpy(argv[0], "Hi!  ");
  ```

[3] `ps`, Process Status, is a Unix command to see what processes are running at the moment.

and then run `ps` while the program `./foo` is still executing, we'll see this instead:

```
4079 tty1     S      0:00 Hi!
```

This behavior is not in the spec and is highly system-dependent.

18.2 Exit Status

Did you notice that the function signatures for `main()` have it returning type `int`? What's that all about? It has to do with a thing called the *exit status*, which is an integer that can be returned to the program that launched yours to let it know how things went.

Now, there are a number of ways a program can exit in C, including `return`ing from `main()`, or calling one of the `exit()` variants.

All of these methods accept an `int` as an argument.

Side note: did you see that in basically all my examples, even though `main()` is supposed to return an `int`, I don't actually `return` anything? In any other function, this would be illegal, but there's a special case in C: if execution reaches the end of `main()` without finding a `return`, it automatically does a `return 0`.

But what does the `0` mean? What other numbers can we put there? And how are they used?

The spec is both clear and vague on the matter, as is common. Clear because it spells out what you can do, but vague in that it doesn't particularly limit it, either.

Nothing for it but to *forge ahead* and figure it out!

Let's get Inception[4] for a second: turns out that when you run your program, *you're running it from another program*.

Usually this other program is some kind of shell[5] that doesn't do much on its own except launch other programs.

But this is a multi-phase process, especially visible in command-line shells:

1. The shell launches your program
2. The shell typically goes to sleep (for command-line shells)
3. Your program runs
4. Your program terminates
5. The shell wakes up and waits for another command

Now, there's a little piece of communication that takes place between steps 4 and 5: the program can return a *status value* that the shell can interrogate. Typically, this value is used to indicate the success or failure of your program, and, if a failure, what type of failure.

This value is what we've been `return`ing from `main()`. That's the status.

Now, the C spec allows for two different status values, which have macro names defined in `<stdlib.h>`:

Status	Description
`EXIT_SUCCESS` or `0`	Program terminated successfully.
`EXIT_FAILURE`	Program terminated with an error.

Let's write a short program that multiplies two numbers from the command line. We'll require that you specify exactly two values. If you don't, we'll print an error message, and exit with an error status.

[4] https://en.wikipedia.org/wiki/Inception
[5] https://en.wikipedia.org/wiki/Shell_(computing)

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

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("usage: mult x y\n");
        return EXIT_FAILURE;   // Indicate to shell that it didn't work
    }

    printf("%d\n", atoi(argv[1]) * atoi(argv[2]));

    return 0;  // same as EXIT_SUCCESS, everything was good.
}
```

Now if we try to run this, we get the expected effect until we specify exactly the right number of command-line arguments:

```
$ ./mult
usage: mult x y

$ ./mult 3 4 5
usage: mult x y

$ ./mult 3 4
12
```

But that doesn't really show the exit status that we returned, does it? We can get the shell to print it out, though. Assuming you're running Bash or another POSIX shell, you can use `echo $?` to see it[6].

Let's try:

```
$ ./mult
usage: mult x y
$ echo $?
1

$ ./mult 3 4 5
usage: mult x y
$ echo $?
1

$ ./mult 3 4
12
$ echo $?
0
```

Interesting! We see that on my system, `EXIT_FAILURE` is `1`. The spec doesn't spell this out, so it could be any number. But try it; it's probably `1` on your system, too.

18.2.1 Other Exit Status Values

The status `0` most definitely means success, but what about all the other integers, even negative ones?

Here we're going off the C spec and into Unix land. In general, while `0` means success, a positive non-zero

[6] In Windows `cmd.exe`, type `echo %errorlevel%`. In PowerShell, type `$LastExitCode`.

number means failure. So you can only have one type of success, and multiple types of failure. Bash says the exit code should be between 0 and 255, though a number of codes are reserved.

In short, if you want to indicate different error exit statuses in a Unix environment, you can start with `1` and work your way up.

On Linux, if you try any code outside the range 0-255, it will bitwise AND the code with `0xff`, effectively clamping it to that range.

You can script the shell to later use these status codes to make decisions about what to do next.

18.3 Environment Variables

Before I get into this, I need to warn you that C doesn't specify what an environment variable is. So I'm going to describe the environment variable system that works on every major platform I'm aware of.

Basically, the environment is the program that's going to run your program, e.g. the bash shell. And it might have some bash variables defined. In case you didn't know, the shell can make its own variables. Each shell is different, but in bash you can just type `set` and it'll show you all of them.

Here's an excerpt from the 61 variables that are defined in my bash shell:

```
HISTFILE=/home/beej/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/home/beej
HOSTNAME=FBILAPTOP
HOSTTYPE=x86_64
IFS=$' \t\n'
```

Notice they are in the form of key/value pairs. For example, one key is `HOSTTYPE` and its value is `x86_64`. From a C perspective, all values are strings, even if they're numbers[7].

So, *anyway*! Long story short, it's possible to get these values from inside your C program.

Let's write a program that uses the standard `getenv()` function to look up a value that you set in the shell.

`getenv()` will return a pointer to the value string, or else `NULL` if the environment variable doesn't exist.

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

int main(void)
{
    char *val = getenv("FROTZ");  // Try to get the value

    // Check to make sure it exists
    if (val == NULL) {
        printf("Cannot find the FROTZ environment variable\n");
        return EXIT_FAILURE;
    }

    printf("Value: %s\n", val);
}
```

If I run this directly, I get this:

[7] If you need a numeric value, convert the string with something like `atoi()` or `strtol()`.

```
$ ./foo
Cannot find the FROTZ environment variable
```

which makes sense, since I haven't set it yet.

In bash, I can set it to something with[8]:

```
$ export FROTZ="C is awesome!"
```

Then if I run it, I get:

```
$ ./foo
Value: C is awesome!
```

In this way, you can set up data in environment variables, and you can get it in your C code and modify your behavior accordingly.

18.3.1 Setting Environment Variables

This isn't standard, but a lot of systems provide ways to set environment variables.

If on a Unix-like, look up the documentation for `putenv()`, `setenv()`, and `unsetenv()`. On Windows, see `_putenv()`.

18.3.2 Unix-like Alternative Environment Variables

If you're on a Unix-like system, odds are you have another couple ways of getting access to environment variables. Note that although the spec points this out as a common extension, it's not truly part of the C standard. It is, however, part of the POSIX standard.

One of these is a variable called `environ` that must be declared like so:

```
extern char **environ;
```

It's an array of strings terminated with a `NULL` pointer.

You should declare it yourself before you use it, or you might find it in the non-standard `<unistd.h>` header file.

Each string is in the form `"key=value"` so you'll have to split it and parse it yourself if you want to get the keys and values out.

Here's an example of looping through and printing out the environment variables a couple different ways:

```
#include <stdio.h>

extern char **environ;  // MUST be extern AND named "environ"

int main(void)
{
    for (char **p = environ; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }
}
```

[8] In Windows CMD.EXE, use `set FROTZ=value`. In PowerShell, use `$Env:FROTZ=value`.

For a bunch of output that looks like this:

```
SHELL=/bin/bash
COLORTERM=truecolor
TERM_PROGRAM_VERSION=1.53.2
LOGNAME=beej
HOME=/home/beej
... etc ...
```

Use `getenv()` if at all possible because it's more portable. But if you have to iterate over environment variables, using `environ` might be the way to go.

Another non-standard way to get the environment variables is as a parameter to `main()`. It works much the same way, but you avoid needing to add your `extern environ` variable. Not even the POSIX spec supports this[9] as far as I can tell, but it's common in Unix land.

```
#include <stdio.h>

int main(int argc, char **argv, char **env)  // <-- env!
{
    (void)argc; (void)argv;  // Suppress unused warnings

    for (char **p = env; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; env[i] != NULL; i++) {
        printf("%s\n", env[i]);
    }
}
```

Just like using `environ` but *even less portable*. It's good to have goals.

[9] https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html

Chapter 19

The C Preprocessor

Before your program gets compiled, it actually runs through a phase called *preprocessing*. It's almost like there's a language *on top* of the C language that runs first. And it outputs the C code, which then gets compiled.

We've already seen this to an extent with `#include`! That's the C Preprocessor! Where it sees that directive, it includes the named file right there, just as if you'd typed it in there. And *then* the compiler builds the whole thing.

But it turns out it's a lot more powerful than just being able to include things. You can define *macros* that are substituted… and even macros that take arguments!

19.1 `#include`

Let's start with the one we've already seen a bunch. This is, of course, a way to include other sources in your source. Very commonly used with header files.

While the spec allows for all kinds of behavior with `#include`, we're going to take a more pragmatic approach and talk about the way it works on every system I've ever seen.

We can split header files into two categories: system and local. Things that are built-in, like `stdio.h`, `stdlib.h`, `math.h`, and so on, you can include with angle brackets:

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

The angle brackets tell C, ``Hey, don't look in the current directory for this header file---look in the system-wide include directory instead."

Which, of course, implies that there must be a way to include local files from the current directory. And there is: with double quotes:

```
#include "myheader.h"
```

Or you can very probably look in relative directories using forward slashes and dots, like this:

```
#include "mydir/myheader.h"
#include "../someheader.py"
```

Don't use a backslash (`\`) for your path separators in your `#include`! It's undefined behavior! Use forward slash (`/`) only, even on Windows.

In summary, used angle brackets (< and >) for the system includes, and use double quotes (") for your personal includes.

19.2 Simple Macros

A *macro* is an identifier that gets *expanded* to another piece of code before the compiler even sees it. Think of it like a placeholder---when the preprocessor sees one of those identifiers, it replaces it with another value that you've defined.

We do this with `#define` (often read ``pound define''). Here's an example:

```
1 #include <stdio.h>
2
3 #define HELLO "Hello, world"
4 #define PI 3.14159
5
6 int main(void)
7 {
8     printf("%s, %f\n", HELLO, PI);
9 }
```

On lines 3 and 4 we defined a couple macros. Wherever these appear elsewhere in the code (line 8), they'll be substituted with the defined values.

From the C compiler's perspective, it's exactly as if we'd written this, instead:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("%s, %f\n", "Hello, world", 3.14159);
6 }
```

See how `HELLO` was replaced with `"Hello, world"` and `PI` was replaced with `3.14159`? From the compiler's perspective, it's just like those values had appeared right there in the code.

Note that the macros don't have a specific type, *per se*. Really all that happens is they get replaced wholesale with whatever they're `#defined` as. If the resulting C code is invalid, the compiler will puke.

You can also define a macro with no value:

```
#define EXTRA_HAPPY
```

in that case, the macro exists and is defined, but is defined to be nothing. So anyplace it occurs in the text will just be replaced with nothing. We'll see a use for this later.

It's conventional to write macro names in `ALL_CAPS` even though that's not technically required.

Overall, this gives you a way to define constant values that are effectively global and can be used *any* place. Even in those places where a `const` variable won't work, e.g. in `switch` `case`s and fixed array lengths.

That said, the debate rages online whether a typed `const` variable is better than `#define` macro in the general case.

It can also be used to replace or modify keywords, a concept completely foreign to `const`, though this practice should be used sparingly.

19.3 Conditional Compilation

It's possible to get the preprocessor to decide whether or not to present certain blocks of code to the compiler, or just remove them entirely before compilation.

We do that by basically wrapping up the code in conditional blocks, similar to `if-else` statements.

19.3.1 If Defined, `#ifdef` and `#endif`

First of all, let's try to compile specific code depending on whether or not a macro is even defined.

```
1  #include <stdio.h>
2
3  #define EXTRA_HAPPY
4
5  int main(void)
6  {
7
8  #ifdef EXTRA_HAPPY
9      printf("I'm extra happy!\n");
10 #endif
11
12     printf("OK!\n");
13 }
```

In that example, we define `EXTRA_HAPPY` (to be nothing, but it *is* defined), then on line 8 we check to see if it is defined with an `#ifdef` directive. If it is defined, the subsequent code will be included up until the `#endif`.

So because it is defined, the code will be included for compilation and the output will be:

```
I'm extra happy!
OK!
```

If we were to comment out the `#define`, like so:

```
//#define EXTRA_HAPPY
```

then it wouldn't be defined, and the code wouldn't be included in compilation. And the output would just be:

```
OK!
```

It's important to remember that these decisions happen at compile time! The code actually gets compiled or removed depending on the condition. This is in contrast to a standard `if` statement that gets evaluated while the program is running.

19.3.2 If Not Defined, `#ifndef`

There's also the negative sense of ``if defined'': ``if not defined'', or `#ifndef`. We could change the previous example to output different things based on whether or not something was defined:

```
8  #ifdef EXTRA_HAPPY
9      printf("I'm extra happy!\n");
10 #endif
11
12 #ifndef EXTRA_HAPPY
13     printf("I'm just regular\n");
14 #endif
```

We'll see a cleaner way to do that in the next section.

Tying it all back in to header files, we've seen how we can cause header files to only be included one time by wrapping them in preprocessor directives like this:

```
#ifndef MYHEADER_H  // First line of myheader.h
#define MYHEADER_H

int x = 12;

#endif  // Last line of myheader.h
```

This demonstrates how a macro persists across files and multiple `#includes`. If it's not yet defined, let's define it and compile the whole header file.

But the next time it's included, we see that `MYHEADER_H` *is* defined, so we don't send the header file to the compiler---it gets effectively removed.

19.3.3 `#else`

But that's not all we can do! There's also an `#else` that we can throw in the mix.

Let's mod the previous example:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#else
    printf("I'm just regular\n");
#endif
```

Now if `EXTRA_HAPPY` is not defined, it'll hit the `#else` clause and print:

```
I'm just regular
```

19.3.4 Else-If: `#elifdef`, `#elifndef`

This feature is new in C23!

What if you want something more complex, though? Perhaps you need an if-else cascade structure to get your code built right?

Luckily we have these directives at our disposal. We can use `#elifdef` for ``else if defined'':

```
#ifdef MODE_1
    printf("This is mode 1\n");
#elifdef MODE_2
    printf("This is mode 2\n");
#elifdef MODE_3
    printf("This is mode 3\n");
#else
    printf("This is some other mode\n");
#endif
```

On the flipside, you can use `#elifndef` for ``else if not defined''.

19.3.5 General Conditional: `#if`, `#elif`

This works very much like the `#ifdef` and `#ifndef` directives in that you can also have an `#else` and the whole thing wraps up with `#endif`.

The only difference is that the constant expression after the `#if` must evaluate to true (non-zero) for the code in the `#if` to be compiled. So instead of whether or not something is defined, we want an expression that evaluates to true.

```
#include <stdio.h>

#define HAPPY_FACTOR 1

int main(void)
{

#if HAPPY_FACTOR == 0
    printf("I'm not happy!\n");
#elif HAPPY_FACTOR == 1
    printf("I'm just regular\n");
#else
    printf("I'm extra happy!\n");
#endif

    printf("OK!\n");
}
```

Again, for the unmatched `#if` clauses, the compiler won't even see those lines. For the above code, after the preprocessor gets finished with it, all the compiler sees is:

```
#include <stdio.h>

int main(void)
{

    printf("I'm just regular\n");

    printf("OK!\n");
}
```

One hackish thing this is used for is to comment out large numbers of lines quickly[1].

If you put an `#if 0` (``if false") at the front of the block to be commented out and an `#endif` at the end, you can get this effect:

```
#if 0
    printf("All this code"); /* is effectively */
    printf("commented out"); // by the #if 0
#endif
```

What if you're on a pre-C23 compiler and you don't have `#elifdef` or `#elifndef` directive support? How can we get the same effect with `#if`? That is, what if I wanted this:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // POTENTIAL ERROR: Not supported before C23
    x = 3;
#endif
```

How could I do it?

Turns out there's a preprocessor operator called `defined` that we can use with an `#if` statement.

[1] You can't always just wrap the code in `/* */` comments because those won't nest.

These are equivalent:

```
#ifdef FOO
#if defined FOO
#if defined(FOO)   // Parentheses optional
```

As are these:

```
#ifndef FOO
#if !defined FOO
#if !defined(FOO)   // Parentheses optional
```

Notice how we can use the standard logical NOT operator (`!`) for ``not defined".

So now we're back in `#if` land and we can use `#elif` with impunity!

This broken code:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // POTENTIAL ERROR: Not supported before C23
    x = 3;
#endif
```

can be replaced with:

```
#if defined FOO
    x = 2;
#elif defined BAR
    x = 3;
#endif
```

19.3.6 Losing a Macro: `#undef`

If you've defined something but you don't need it any longer, you can undefine it with `#undef`.

```
#include <stdio.h>

int main(void)
{
#define GOATS

#ifdef GOATS
    printf("Goats detected!\n");  // prints
#endif

#undef GOATS  // Make GOATS no longer defined

#ifdef GOATS
    printf("Goats detected, again!\n"); // doesn't print
#endif
}
```

19.4 Built-in Macros

The standard defines a lot of built-in macros that you can test and use for conditional compilation. Let's look at those here.

19.4.1 Mandatory Macros

These are all defined:

Macro	Description
`__DATE__`	The date of compilation---like when you're compiling this file---in `Mmm dd yyyy` format
`__TIME__`	The time of compilation in `hh:mm:ss` format
`__FILE__`	A string containing this file's name
`__LINE__`	The line number of the file this macro appears on
`__func__`	The name of the function this appears in, as a string[2]
`__STDC__`	Defined with `1` if this is a standard C compiler
`__STDC_HOSTED__`	This will be `1` if the compiler is a *hosted implementation*[3], otherwise `0`
`__STDC_VERSION__`	This version of C, a constant `long int` in the form `yyyymmL`, e.g. `201710L`

Let's put these together.

```
#include <stdio.h>

int main(void)
{
    printf("This function: %s\n", __func__);
    printf("This file: %s\n", __FILE__);
    printf("This line: %d\n", __LINE__);
    printf("Compiled on: %s %s\n", __DATE__, __TIME__);
    printf("C Version: %ld\n", __STDC_VERSION__);
}
```

The output on my system is:

```
This function: main
This file: foo.c
This line: 7
Compiled on: Nov 23 2020 17:16:27
C Version: 201710
```

`__FILE__`, `__func__` and `__LINE__` are particularly useful to report error conditions in messages to developers. The `assert()` macro in `<assert.h>` uses these to call out where in the code the assertion failed.

19.4.1.1 `__STDC_VERSION__`s

In case you're wondering, here are the version numbers for different major releases of the C Language Spec:

Release	ISO/IEC version	`__STDC_VERSION__`
C89	ISO/IEC 9899:1990	undefined
C89	ISO/IEC 9899:1990/Amd.1:1995	`199409L`
C99	ISO/IEC 9899:1999	`199901L`
C11	ISO/IEC 9899:2011/Amd.1:2012	`201112L`

[2]This isn't really a macro---it's technically an identifier. But it's the only predefined identifier and it feels very macro-like, so I'm including it here. Like a rebel.

[3]A hosted implementation basically means you're running the full C standard, probably on an operating system of some kind. Which you probably are. If you're running on bare metal in some kind of embedded system, you're probably on a *standalone implementation*.

Note the macro did not exist originally in C89.

Also note that the plan is that the version numbers will strictly increase, so you could always check for, say, ``at least C99'' with:

```
#if __STDC_VERSION__ >= 1999901L
```

19.4.2 Optional Macros

Your implementation might define these, as well. Or it might not.

Macro	Description
`__STDC_ISO_10646__`	If defined, `wchar_t` holds Unicode values, otherwise something else
`__STDC_MB_MIGHT_NEQ_WC__`	A `1` indicates that the values in multibyte characters might not map equally to values in wide characters
`__STDC_UTF_16__`	A `1` indicates that the system uses UTF-16 encoding in type `char16_t`
`__STDC_UTF_32__`	A `1` indicates that the system uses UTF-32 encoding in type `char32_t`
`__STDC_ANALYZABLE__`	A `1` indicates the code is analyzable[4]
`__STDC_IEC_559__`	`1` if IEEE-754 (aka IEC 60559) floating point is supported
`__STDC_IEC_559_COMPLEX__`	`1` if IEC 60559 complex floating point is supported
`__STDC_LIB_EXT1__`	`1` if this implementation supports a variety of ``safe'' alternate standard library functions (they have `_s` suffixes on the name)
`__STDC_NO_ATOMICS__`	`1` if this implementation does **not** support `_Atomic` or `<stdatomic.h>`
`__STDC_NO_COMPLEX__`	`1` if this implementation does **not** support complex types or `<complex.h>`
`__STDC_NO_THREADS__`	`1` if this implementation does **not** support `<threads.h>`
`__STDC_NO_VLA__`	`1` if this implementation does **not** support variable-length arrays

19.5 Macros with Arguments

Macros are more powerful than simple substitution, though. You can set them up to take arguments that are substituted in, as well.

A question often arises for when to use parameterized macros versus functions. Short answer: use functions. But you'll see lots of macros in the wild and in the standard library. People tend to use them for short, mathy things, and also for features that might change from platform to platform. You can define different keywords for one platform or another.

19.5.1 Macros with One Argument

Let's start with a simple one that squares a number:

[4] OK, I know that was a cop-out answer. Basically there's an optional extension compilers can implement wherein they agree to limit certain types of undefined behavior so that the C code is more amenable to static code analysis. It is unlikely you'll need to use this.

```
1  #include <stdio.h>
2
3  #define SQR(x) x * x  // Not quite right, but bear with me
4
5  int main(void)
6  {
7      printf("%d\n", SQR(12));  // 144
8  }
```

What that's saying is ``everywhere you see `SQR` with some value, replace it with that value times itself''.

So line 7 will be changed to:

```
7      printf("%d\n", 12 * 12);  // 144
```

which C comfortably converts to 144.

But we've made an elementary error in that macro, one that we need to avoid.

Let's check it out. What if we wanted to compute `SQR(3 + 4)`? Well, $3 + 4 = 7$, so we must want to compute $7^2 = 49$. That's it; `49`---final answer.

Let's drop it in our code and see that we get… 19?

```
7      printf("%d\n", SQR(3 + 4));  // 19!!??
```

What happened?

If we follow the macro expansion, we get

```
7      printf("%d\n", 3 + 4 * 3 + 4);  // 19!
```

Oops! Since multiplication takes precedence, we do the $4 \times 3 = 12$ first, and get $3 + 12 + 4 = 19$. Not what we were after.

So we have to fix this to make it right.

This is so common that you should automatically do it every time you make a parameterized math macro!

The fix is easy: just add some parentheses!

```
3  #define SQR(x) (x) * (x)   // Better... but still not quite good enough!
```

And now our macro expands to:

```
7      printf("%d\n", (3 + 4) * (3 + 4));  // 49! Woo hoo!
```

But we actually still have the same problem which might manifest if we have a higher-precedence operator than multiply (`*`) nearby.

So the safe, proper way to put the macro together is to wrap the whole thing in additional parentheses, like so:

```
3  #define SQR(x) ((x) * (x))   // Good!
```

Just make it a habit to do that when you make a math macro and you can't go wrong.

19.5.2 Macros with More than One Argument

You can stack these things up as much as you want:

```
#define TRIANGLE_AREA(w, h) (0.5 * (w) * (h))
```

Let's do some macros that solve for x using the quadratic formula. Just in case you don't have it on the top of your head, it says for equations of the form:

$$ax^2 + bx + c = 0$$

you can solve for x with the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Which is crazy. Also notice the plus-or-minus ($\pm$) in there, indicating that there are actually two solutions.

So let's make macros for both:

```
#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
```

So that gets us some math. But let's define one more that we can use as arguments to `printf()` to print both answers.

```
//          macro              replacement
//      |-----------| |---------------------------|
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
```

That's just a couple values separated by a comma---and we can use that as a ``combined" argument of sorts to `printf()` like this:

```
printf("x = %f or x = %f\n", QUAD(2, 10, 5));
```

Let's put it together into some code:

```
#include <stdio.h>
#include <math.h>  // For sqrt()

#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)

int main(void)
{
    printf("2*x^2 + 10*x + 5 = 0\n");
    printf("x = %f or x = %f\n", QUAD(2, 10, 5));
}
```

And this gives us the output:

```
2*x^2 + 10*x + 5 = 0
x = -0.563508 or x = -4.436492
```

Plugging in either of those values gives us roughly zero (a bit off because the numbers aren't exact):

$$2 \times -0.563508^2 + 10 \times -0.563508 + 5 \approx 0.000003$$

19.5.3 Macros with Variable Arguments

There's also a way to have a variable number of arguments passed to a macro, using ellipses (`...`) after the known, named arguments. When the macro is expanded, all of the extra arguments will be in a comma-separated list in the `__VA_ARGS__` macro, and can be replaced from there:

```
#include <stdio.h>

// Combine the first two arguments to a single number,
```

```
4  // then have a commalist of the rest of them:
5
6  #define X(a, b, ...) (10*(a) + 20*(b)), __VA_ARGS__
7
8  int main(void)
9  {
10     printf("%d %f %s %d\n", X(5, 4, 3.14, "Hi!", 12));
11 }
```

The substitution that takes place on line 10 would be:

```
10     printf("%d %f %s %d\n", (10*(5) + 20*(4)), 3.14, "Hi!", 12);
```

for output:

```
130 3.140000 Hi! 12
```

You can also ``stringify" __VA_ARGS__ by putting a # in front of it:

```
#define X(...) #__VA_ARGS__

printf("%s\n", X(1,2,3));  // Prints "1, 2, 3"
```

19.5.4 Stringification

Already mentioned, just above, you can turn any argument into a string by preceding it with a # in the replacement text.

For example, we could print anything as a string with this macro and `printf()`:

```
#define STR(x) #x

printf("%s\n", STR(3.14159));
```

In that case, the substitution leads to:

```
printf("%s\n", "3.14159");
```

Let's see if we can use this to greater effect so that we can pass any `int` variable name into a macro, and have it print out it's name and value.

```
1  #include <stdio.h>
2
3  #define PRINT_INT_VAL(x) printf("%s = %d\n", #x, x)
4
5  int main(void)
6  {
7      int a = 5;
8
9      PRINT_INT_VAL(a);  // prints "a = 5"
10 }
```

On line 9, we get the following macro replacement:

```
9      printf("%s = %d\n", "a", 5);
```

19.5.5 Concatenation

We can concatenate two arguments together with ##, as well. Fun times!

```
#define CAT(a, b) a ## b

printf("%f\n", CAT(3.14, 1592));   // 3.141592
```

19.6 Multiline Macros

It's possible to continue a macro to multiple lines if you escape the newline with a backslash (\).

Let's write a multiline macro that prints numbers from `0` to the product of the two arguments passed in.

```
#include <stdio.h>

#define PRINT_NUMS_TO_PRODUCT(a, b) do { \
    int product = (a) * (b); \
    for (int i = 0; i < product; i++) { \
        printf("%d\n", i); \
    } \
} while(0)

int main(void)
{
    PRINT_NUMS_TO_PRODUCT(2, 4);  // Outputs numbers from 0 to 7
}
```

A couple things to note there:

- Escapes at the end of every line except the last one to indicate that the macro continues.
- The whole thing is wrapped in a `do-while(0)` loop with squirrley braces.

The latter point might be a little weird, but it's all about absorbing the trailing `;` the coder drops after the macro.

At first I thought that just using squirrely braces would be enough, but there's a case where it fails if the coder puts a semicolon after the macro. Here's that case:

```
#include <stdio.h>

#define FOO(x) { (x)++; }

int main(void)
{
    int i = 0;

    if (i == 0)
        FOO(i);
    else
        printf(":-(\n");

    printf("%d\n", i);
}
```

Looks simple enough, but it won't build without a syntax error:

```
foo.c:11:5: error: ‘else’ without a previous ‘if’
```

Do you see it?

Let's look at the expansion:

```
    if (i == 0) {
        (i)++;
    };              // <-- Trouble with a capital-T!

    else
        printf(":-(\n");
```

The `;` puts an end to the `if` statement, so the `else` is just floating out there illegally[5].

So wrap that multiline macro with a `do-while(0)`.

19.7 Example: An Assert Macro

Adding asserts to your code is a good way to catch conditions that you think shouldn't happen. C provides `assert()` functionality. It checks a condition, and if it's false, the program bombs out telling you the file and line number on which the assertion failed.

But this is wanting.

1. First of all, you can't specify an additional message with the assert.
2. Secondly, there's no easy on-off switch for all the asserts.

We can address the first with macros.

Basically, when I have this code:

```
ASSERT(x < 20, "x must be under 20");
```

I want something like this to happen (assuming the `ASSERT()` is on line 220 of `foo.c`):

```
if (!(x < 20)) {
    fprintf(stderr, "foo.c:220: assertion x < 20 failed: ");
    fprintf(stderr, "x must be under 20\n");
    exit(1);
}
```

We can get the filename out of the `__FILE__` macro, and the line number from `__LINE__`. The message is already a string, but `x < 20` is not, so we'll have to stringify it with `#`. We can make a multiline macro by using backslash escapes at the end of the line.

```
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
```

(It looks a little weird with `__FILE__` out front like that, but remember it is a string literal, and string literals next to each other are automagically concatenated. `__LINE__` on the other hand, it's just an `int`.)

And that works! If I run this:

```
int x = 30;

ASSERT(x < 20, "x must be under 20");
```

[5] *Breakin' the law… breakin' the law…*

I get this output:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

Very nice!

The only thing left is a way to turn it on and off, and we could do that with conditional compilation.

Here's the complete example:

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

#define ASSERT_ENABLED 1

#if ASSERT_ENABLED
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
#else
#define ASSERT(c, m)  // Empty macro if not enabled
#endif

int main(void)
{
    int x = 30;

    ASSERT(x < 20, "x must be under 20");
}
```

This has the output:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

19.8 The `#error` Directive

This directive causes the compiler to error out as soon as it sees it.

Commonly, this is used inside a conditional to prevent compilation unless some prerequisites are met:

```
#ifndef __STDC_IEC_559__
    #error I really need IEEE-754 floating point to compile. Sorry!
#endif
```

Some compilers have a non-standard complementary `#warning` directive that will output a warning but not stop compilation, but this is not in the C11 spec.

19.9 The `#embed` Directive

New in C23!

And currently not yet working with any of my compilers, so take this section with a grain of salt!

The gist of this is that you can include bytes of a file as integer constants as if you'd typed them in.

For example, if you have a binary file named `foo.bin` that contains four bytes with decimal values 11, 22, 33, and 44, and you do this:

```
int a[] = {
#embed "foo.bin"
};
```

It'll be just as if you'd typed this:

```
int a[] = {11,22,33,44};
```

This is a really powerful way to initialize an array with binary data without needing to convert it all to code first---the preprocessor does it for you!

A more typical use case might be a file containing a small image to be displayed that you don't want to load at runtime.

Here's another example:

```
int a[] = {
#embed <foo.bin>
};
```

If you use angle brackets, the preprocessor looks in a series of implementation-defined places to locate the file, just like `#include` would do. If you use double quotes and the resource is not found, the compiler will try it as if you'd used angle brackets in a last desperate attempt to find the file.

`#embed` works like `#include` in that it effectively pastes values in before the compiler sees them. This means you can use it in all kinds of places:

```
return
#embed "somevalue.dat"
;
```

or

```
int x =
#embed "xvalue.dat"
;
```

Now---are these always bytes? Meaning they'll have values from 0 to 255, inclusive? The answer is definitely by default ``yes'', except when it is ``no''.

Technically, the elements will be `CHAR_BIT` bits wide. And this is very likely 8 on your system, so you'd get that 0-255 range in your values. (They'll always be non-negative.)

Also, it's possible that an implementation might allow this to be overridden in some way, e.g. on the command line or with parameters.

The size of the file in bits must be a multiple of the element size. That is, if each element is 8 bits, the file size (in bits) must be a multiple of 8. In regular everyday usage, this is a confusing way of saying that each file needs to be an integer number of bytes… which of course it is. Honestly, I'm not even sure why I bothered with this paragraph. Read the spec if you're really that curious.

19.9.1 `#embed` Parameters

There are all kinds of parameters you can specify to the `#embed` directive. Here's an example with the yet-unintroduced `limit()` parameter:

```
int a[] = {
#embed "/dev/random" limit(5)
};
```

But what if you already have `limit` defined somewhere else? Luckily you can put __ around the keyword and it will work the same way:

```
int a[] = {
#embed "/dev/random" __limit__(5)
};
```

Now… what's this `limit` thing?

19.9.2 The `limit()` Parameter

You can specify a limit on the number of elements to embed with this parameter.

This is a maximum value, not an absolute value. If the file that's embedded is shorter than the specified limit, only that many bytes will be imported.

The `/dev/random` example above is an example of the motivation for this---in Unix, that's a *character device file* that will return an infinite stream of pretty-random numbers.

Embedding an infinite number of bytes is hard on your RAM, so the `limit` parameter gives you a way to stop after a certain number.

Finally, you are allowed to use `#define` macros in your `limit`, in case you were curious.

19.9.3 The `if_empty` Parameter

This parameter defines what the embed result should be if the file exists but contains no data. Let's say that the file `foo.dat` contains a single byte with the value 123. If we do this:

```
int x =
#embed "foo.dat" if_empty(999)
;
```

we'll get:

```
int x = 123;   // When foo.dat contains a 123 byte
```

But what if the file `foo.dat` is zero bytes long (i.e. contains no data and is empty)? If that's the case, it would expand to:

```
int x = 999;   // When foo.dat is empty
```

Notably if the `limit` is set to `0`, then the `if_empty` will always be substituted. That is, a zero limit effectively means the file is empty.

This will always emit `x = 999` no matter what's in `foo.dat`:

```
int x =
#embed "foo.dat" limit(0) if_empty(999)
;
```

19.9.4 The `prefix()` and `suffix()` Parameters

This is a way to prepend some data on the embed.

Note that these only affect non-empty data! If the file is empty, neither `prefix` nor `suffix` has any effect.

Here's an example where we embed three random numbers, but prefix the numbers with `11,` and suffix them with `,99`:

```
int x[] = {
#embed "/dev/urandom" limit(3) prefix(11,) suffix(,99)
};
```

Example result:

```
int x[] = {11,135,116,220,99};
```

There's no requirement that you use both `prefix` and `suffix`. You can use both, one, the other, or neither.

We can make use of the characteristic that these are only applied to non-empty files to neat effect, as shown in the following example shamelessly stolen from the spec.

Let's say we have a file `foo.dat` that has some data it in. And we want to use this to initialize an array, and then we want a suffix on the array that is a zero element.

No problem, right?

```
int x[] = {
#embed "foo.dat" suffix(,0)
};
```

If `foo.dat` has 11, 22, and 33 in it, we'd get:

```
int x[] = {11,22,33,0};
```

But wait! What if `foo.dat` is empty? Then we get:

```
int x[] = {};
```

and that's not good.

But we can fix it like this:

```
int x[] = {
#embed "foo.dat" suffix(,)
    0
};
```

Since the `suffix` parameter is omitted if the file is empty, this would just turn into:

```
int x[] = {0};
```

which is fine.

19.9.5 The `__has_embed()` Identifier

This is a great way to test to see if a particular file is available to be embedded, and also whether or not it's empty.

You use it with the `#if` directive.

Here's a chunk of code that will get 5 random numbers from the random number generator character device. If that doesn't exist, it tries to get them from a file `myrandoms.dat`. If that doesn't exist, it uses some hard-coded values:

```
    int random_nums[] = {
#if __has_embed("/dev/urandom")
    #embed "/dev/urandom" limit(5)
#elif __has_embed("myrandoms.dat")
```

```
    #embed "myrandoms.dat" limit(5)
#else
    140,178,92,167,120
#endif
    };
```

Technically, the `__has_embed()` identifier resolves to one of three values:

`__has_embed()` Result	Description
`__STDC_EMBED_NOT_FOUND__`	If the file isn't found.
`__STDC_EMBED_FOUND__`	If the file is found and is not empty.
`__STDC_EMBED_EMPTY`	If the file is found and is empty.

I have good reason to believe that `__STDC_EMBED_NOT_FOUND__` is `0` and the others aren't zero (because it's implied in the proposal and it makes logical sense), but I'm having trouble finding that in this version of the draft spec.

TODO

19.9.6 Other Parameters

A compiler implementation can define other embed parameters all it wants---look for these non-standard parameters in your compiler's documentation.

For instance:

```
#embed "foo.bin" limit(12) frotz(lamp)
```

These might commonly have a prefix on them to help with namespacing:

```
#embed "foo.bin" limit(12) fmc::frotz(lamp)
```

It might be sensible to try to detect if these are available before you use them, and luckily we can use `__has_embed` to help us here.

Normally, `__has_embed()` will just tell us if the file is there or not. But---and here's the fun bit---it will also return false if any additional parameters are also not supported!

So if we give it a file that we *know* exists as well as a parameter that we want to test for the existence of, it will effectively tell us if that parameter is supported.

What file *always* exists, though? Turns out we can use the `__FILE__` macro, which expands to the name of the source file that references it! That file *must* exist, or something is seriously wrong in the chicken-and-egg department.

Let's test that `frotz` parameter to see if we can use it:

```
#if __has_embed(__FILE__ fmc::frotz(lamp))
    puts("fmc::frotz(lamp) is supported!");
#else
    puts("fmc::frotz(lamp) is NOT supported!");
#endif
```

19.9.7 Embedding Multi-Byte Values

What about getting some `int`s in there instead of individual bytes? What about multi-byte values in the embedded file?

This is not something supported by the C23 standard, but there could be implementation extensions defined for it in the future.

19.10 The `#pragma` Directive

This is one funky directive, short for ``pragmatic''. You can use it to do… well, anything your compiler supports you doing with it.

Basically the only time you're going to add this to your code is if some documentation tells you to do so.

19.10.1 Non-Standard Pragmas

Here's one non-standard example of using `#pragma` to cause the compiler to execute a `for` loop in parallel with multiple threads (if the compiler supports the OpenMP[6] extension):

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) { ... }
```

There are all kinds of `#pragma` directives documented across all four corners of the globe.

All unrecognized `#pragma`s are ignored by the compiler.

19.10.2 Standard Pragmas

There are also a few standard ones, and these start with `STDC`, and follow the same form:

```
#pragma STDC pragma_name on-off
```

The `on-off` portion can be either `ON`, `OFF`, or `DEFAULT`.

And the `pragma_name` can be one of these:

Pragma Name	Description
`FP_CONTRACT`	Allow floating point expressions to be contracted into a single operation to avoid rounding errors that might occur from multiple operations.
`FENV_ACCESS`	Set to `ON` if you plan to access the floating point status flags. If `OFF`, the compiler might perform optimizations that cause the values in the flags to be inconsistent or invalid.
`CX_LIMITED_RANGE`	Set to `ON` to allow the compiler to skip overflow checks when performing complex arithmetic. Defaults to `OFF`.

For example:

```
#pragma STDC FP_CONTRACT OFF
#pragma STDC CX_LIMITED_RANGE ON
```

As for `CX_LIMITED_RANGE`, the spec points out:

> The purpose of the pragma is to allow the implementation to use the formulas:
>
> $(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$
>
> $(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$

[6] https://www.openmp.org/

$$|x + iy| = \sqrt{x^2 + y^2}$$

where the programmer can determine they are safe.

19.10.3 `_Pragma` Operator

This is another way to declare a pragma that you could use in a macro.

These are equivalent:

```
#pragma "Unnecessary" quotes
_Pragma("\"Unnecessary\" quotes")
```

This can be used in a macro, if need be:

```
#define PRAGMA(x) _Pragma(#x)
```

19.11 The `#line` Directive

This allows you to override the values for `__LINE__` and `__FILE__`. If you want.

I've never wanted to do this, but in K&R2, they write:

> For the benefit of other preprocessors that generate C programs […]

So maybe there's that.

To override the line number to, say 300:

```
#line 300
```

and `__LINE__` will keep counting up from there.

To override the line number and the filename:

```
#line 300 "newfilename"
```

19.12 The Null Directive

A # on a line by itself is ignored by the preprocessor. Now, to be entirely honest, I don't know what the use case is for this.

I've seen examples like this:

```
#ifdef FOO
    #
#else
    printf("Something");
#endif
```

which is just cosmetic; the line with the solitary # can be deleted with no ill effect.

Or maybe for cosmetic consistency, like this:

```
#
#ifdef FOO
    x = 2;
#endif
#
#if BAR == 17
    x = 12;
```

```
#endif
#
```

But, with respect to cosmetics, that's just ugly.

Another post mentions elimination of comments---that in GCC, a comment after a # will not be seen by the compiler. Which I don't doubt, but the specification doesn't seem to say this is standard behavior.

My searches for rationale aren't bearing much fruit. So I'm going to just say this is some good ol' fashioned C esoterica.

Chapter 20

`structs` II: More Fun with `structs`

Turns out there's a lot more you can do with `structs` than we've talked about, but it's just a big pile of miscellaneous things. So we'll throw them in this chapter.

If you're good with `struct` basics, you can round out your knowledge here.

20.1 Initializers of Nested `structs` and Arrays

Remember how you could initialize structure members along these lines?

```
struct foo x = {.a=12, .b=3.14};
```

Turns out we have more power in these initializers than we'd originally shared. Exciting!

For one thing, if you have a nested substructure like the following, you can initialize members of that substructure by following the variable names down the line:

```
struct foo x = {.a.b.c=12};
```

Let's look at an example:

```
#include <stdio.h>

struct cabin_information {
    int window_count;
    int o2level;
};

struct spaceship {
    char *manufacturer;
    struct cabin_information ci;
};

int main(void)
{
    struct spaceship s = {
        .manufacturer="General Products",
        .ci.window_count = 8,   // <-- NESTED INITIALIZER!
        .ci.o2level = 21
    };

```

```
21      printf("%s: %d seats, %d%% oxygen\n",
22          s.manufacturer, s.ci.window_count, s.ci.o2level);
23  }
```

Check out lines 16-17! That's where we're initializing members of the `struct cabin_information` in the definition of `s`, our `struct spaceship`.

And here is another option for that same initializer---this time we'll do something more standard-looking, but either approach works:

```
15      struct spaceship s = {
16          .manufacturer="General Products",
17          .ci={
18              .window_count = 8,
19              .o2level = 21
20          }
21      };
```

Now, as if the above information isn't spectacular enough, we can also mix in array initializers in there, too.

Let's change this up to get an array of passenger information in there, and we can check out how the initializers work in there, too.

```
1   #include <stdio.h>
2
3   struct passenger {
4       char *name;
5       int covid_vaccinated; // Boolean
6   };
7
8   #define MAX_PASSENGERS 8
9
10  struct spaceship {
11      char *manufacturer;
12      struct passenger passenger[MAX_PASSENGERS];
13  };
14
15  int main(void)
16  {
17      struct spaceship s = {
18          .manufacturer="General Products",
19          .passenger = {
20              // Initialize a field at a time
21              [0].name = "Gridley, Lewis",
22              [0].covid_vaccinated = 0,
23
24              // Or all at once
25              [7] = {.name="Brown, Teela", .covid_vaccinated=1},
26          }
27      };
28
29      printf("Passengers for %s ship:\n", s.manufacturer);
30
31      for (int i = 0; i < MAX_PASSENGERS; i++)
32          if (s.passenger[i].name != NULL)
33              printf("    %s (%svaccinated)\n",
```

```
34                  s.passenger[i].name,
35                  s.passenger[i].covid_vaccinated? "": "not ");
36  }
```

20.2 Anonymous structs

These are ``the `struct` with no name''. We also mention these in the `typedef` section, but we'll refresh here.

Here's a regular `struct`:

```
struct animal {
    char *name;
    int leg_count, speed;
};
```

And here's the anonymous equivalent:

```
struct {              // <-- No name!
    char *name;
    int leg_count, speed;
};
```

Okaaaaay. So we have a `struct`, but it has no name, so we have no way of using it later? Seems pretty pointless.

Admittedly, in that example, it is. But we can still make use of it a couple ways.

One is rare, but since the anonymous `struct` represents a type, we can just put some variable names after it and use them.

```
struct {                    // <-- No name!
    char *name;
    int leg_count, speed;
} a, b, c;                  // 3 variables of this struct type

a.name = "antelope";
c.leg_count = 4;            // for example
```

But that's still not that useful.

Far more common is use of anonymous `structs` with a `typedef` so that we can use it later (e.g. to pass variables to functions).

```
typedef struct {                    // <-- No name!
    char *name;
    int leg_count, speed;
} animal;                           // New type: animal

animal a, b, c;

a.name = "antelope";
c.leg_count = 4;            // for example
```

Personally, I don't use many anonymous `structs`. I think it's more pleasant to see the entire `struct animal` before the variable name in a declaration.

But that's just, like, my opinion, man.

20.3 Self-Referential `structs`

For any graph-like data structure, it's useful to be able to have pointers to the connected nodes/vertices. But this means that in the definition of a node, you need to have a pointer to a node. It's chicken and eggy!

But it turns out you can do this in C with no problem whatsoever.

For example, here's a linked list node:

```
struct node {
    int data;
    struct node *next;
};
```

It's important to note that `next` is a pointer. This is what allows the whole thing to even build. Even though the compiler doesn't know what the entire `struct node` looks like yet, all pointers are the same size.

Here's a cheesy linked list program to test it out:

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

struct node {
    int data;
    struct node *next;
};

int main(void)
{
    struct node *head;

    // Hackishly set up a linked list (11)->(22)->(33)
    head = malloc(sizeof(struct node));
    head->data = 11;
    head->next = malloc(sizeof(struct node));
    head->next->data = 22;
    head->next->next = malloc(sizeof(struct node));
    head->next->next->data = 33;
    head->next->next->next = NULL;

    // Traverse it
    for (struct node *cur = head; cur != NULL; cur = cur->next) {
        printf("%d\n", cur->data);
    }
}
```

Running that prints:

```
11
22
33
```

20.4 Flexible Array Members

Back in the good old days, when people carved C code out of wood, some folks thought would be neat if they could allocate `structs` that had variable length arrays at the end of them.

I want to be clear that the first part of the section is the old way of doing things, and we're going to do things the new way after that.

For example, maybe you could define a `struct` for holding strings and the length of that string. It would have a length and an array to hold the data. Maybe something like this:

```
struct len_string {
    int length;
    char data[8];
};
```

But that has `8` hardcoded as the maximum length of a string, and that's not much. What if we did something *clever* and just `malloc()`d some extra space at the end after the struct, and then let the data overflow into that space?

Let's do that, and then allocate another 40 bytes on top of it:

```
struct len_string *s = malloc(sizeof *s + 40);
```

Because `data` is the last field of the `struct`, if we overflow that field, it runs out into space that we already allocated! For this reason, this trick only works if the short array is the *last* field in the `struct`.

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");  // Won't crash. Probably.
```

In fact, there was a common compiler workaround for doing this, where you'd allocate a zero length array at the end:

```
struct len_string {
    int length;
    char data[0];
};
```

And then every extra byte you allocated was ready for use in that string.

Because `data` is the last field of the `struct`, if we overflow that field, it runs out into space that we already allocated!

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");  // Won't crash. Probably.
```

But, of course, actually accessing the data beyond the end of that array is undefined behavior! In these modern times, we no longer deign to resort to such savagery.

Luckily for us, we can still get the same effect with C99 and later, but now it's legal.

Let's just change our above definition to have no size for the array[1]:

```
struct len_string {
    int length;
    char data[];
};
```

Again, this only works if the flexible array member is the *last* field in the `struct`.

And then we can allocate all the space we want for those strings by `malloc()`ing larger than the `struct len_string`, as we do in this example that makes a new `struct len_string` from a C string:

[1]Technically we say that it has an *incomplete type*.

```
struct len_string *len_string_from_c_string(char *s)
{
    int len = strlen(s);

    // Allocate "len" more bytes than we'd normally need
    struct len_string *ls = malloc(sizeof *ls + len);

    ls->length = len;

    // Copy the string into those extra bytes
    memcpy(ls->data, s, len);

    return ls;
}
```

20.5 Padding Bytes

Beware that C is allowed to add padding bytes within or after a `struct` as it sees fit. You can't trust that they will be directly adjacent in memory[2].

Let's take a look at this program. We output two numbers. One is the sum of the `sizeof`s the individual field types. The other is the `sizeof` the entire `struct`.

One would expect them to be the same. The size of the total is the size of the sum of its parts, right?

```
#include <stdio.h>

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", sizeof(int) + sizeof(char) + sizeof(int) + sizeof(char));
    printf("%zu\n", sizeof(struct foo));
}
```

But on my system, this outputs:

```
10
16
```

They're not the same! The compiler has added 6 bytes of padding to help it be more performant. Maybe you got different output with your compiler, but unless you're forcing it, you can't be sure there's no padding.

20.6 `offsetof`

In the previous section, we saw that the compiler could inject padding bytes at will inside a structure.

What if we needed to know where those were? We can measure it with `offsetof`, defined in `<stddef.h>`.

[2]Though some compilers have options to force this to occur---search for `__attribute__((packed))` to see how to do this with GCC.

Let's modify the code from above to print the offsets of the individual fields in the `struct`:

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

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", offsetof(struct foo, a));
    printf("%zu\n", offsetof(struct foo, b));
    printf("%zu\n", offsetof(struct foo, c));
    printf("%zu\n", offsetof(struct foo, d));
}
```

For me, this outputs:

```
0
4
8
12
```

indicating that we're using 4 bytes for each of the fields. It's a little weird, because `char` is only 1 byte, right? The compiler is putting 3 padding bytes after each `char` so that all the fields are 4 bytes long. Presumably this will run faster on my CPU.

20.7 Fake OOP

There's a slightly abusive thing that's sort of OOP-like that you can do with `structs`.

Since the pointer to the `struct` is the same as a pointer to the first element of the `struct`, you can freely cast a pointer to the `struct` to a pointer to the first element.

What this means is that we can set up a situation like this:

```
struct parent {
    int a, b;
};

struct child {
    struct parent super;  // MUST be first
    int c, d;
};
```

Then we are able to pass a pointer to a `struct child` to a function that expects either that *or* a pointer to a `struct parent`!

Because `struct parent super` is the first item in the `struct child`, a pointer to any `struct child` is the same as a pointer to that `super` field[3].

[3] `super` isn't a keyword, incidentally. I'm just stealing some OOP terminology.

Let's set up an example here. We'll make `structs` as above, but then we'll pass a pointer to a `struct child` to a function that needs a pointer to a `struct parent`… and it'll still work.

```
#include <stdio.h>

struct parent {
    int a, b;
};

struct child {
    struct parent super;  // MUST be first
    int c, d;
};

// Making the argument `void*` so we can pass any type into it
// (namely a struct parent or struct child)
void print_parent(void *p)
{
    // Expects a struct parent--but a struct child will also work
    // because the pointer points to the struct parent in the first
    // field:
    struct parent *self = p;

    printf("Parent: %d, %d\n", self->a, self->b);
}

void print_child(struct child *self)
{
    printf("Child: %d, %d\n", self->c, self->d);
}

int main(void)
{
    struct child c = {.super.a=1, .super.b=2, .c=3, .d=4};

    print_child(&c);
    print_parent(&c);  // Also works even though it's a struct child!
}
```

See what we did on the last line of `main()`? We called `print_parent()` but passed a `struct child*` as the argument! Even though `print_parent()` needs the argument to point to a `struct parent`, we're *getting away with it* because the first field in the `struct child` is a `struct parent`.

Again, this works because a pointer to a `struct` has the same value as a pointer to the first field in that `struct`.

This all hinges on this part of the spec:

> **§6.7.2.1¶15** […] A pointer to a structure object, suitably converted, points to its initial member […], and vice versa.

and

> **§6.5¶7** An object shall have its stored value accessed only by an lvalue expression that has one of the following types:
>
> - a type compatible with the effective type of the object
> - […]

and my assumption that ``suitably converted" means ``cast to the effective type of the initial member".

20.8 Bit-Fields

In my experience, these are rarely used, but you might see them out there from time to time, especially in lower-level applications that pack bits together into larger spaces.

Let's take a look at some code to demonstrate a use case:

```
#include <stdio.h>

struct foo {
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
};

int main(void)
{
    printf("%zu\n", sizeof(struct foo));
}
```

For me, this prints `16`. Which makes sense, since `unsigned`s are 4 bytes on my system.

But what if we knew that all the values that were going to be stored in `a` and `b` could be stored in 5 bits, and the values in `c`, and `d` could be stored in 3 bits? That's only a total 16 bits. Why have 128 bits reserved for them if we're only going to use 16?

Well, we can tell C to pretty-please try to pack these values in. We can specify the maximum number of bits that values can take (from 1 up the size of the containing type).

We do this by putting a colon after the field name, followed by the field width in bits.

```
struct foo {
    unsigned int a:5;
    unsigned int b:5;
    unsigned int c:3;
    unsigned int d:3;
};
```

Now when I ask C how big my `struct foo` is, it tells me 4! It was 16 bytes, but now it's only 4. It has ``packed" those 4 values down into 4 bytes, which is a four-fold memory savings.

The tradeoff is, of course, that the 5-bit fields can only hold values from 0-31 and the 3-bit fields can only hold values from 0-7. But life's all about compromise, after all.

20.8.1 Non-Adjacent Bit-Fields

A gotcha: C will only combine **adjacent** bit-fields. If they're interrupted by non-bit-fields, you get no savings:

```
struct foo {            // sizeof(struct foo) == 16 (for me)
    unsigned int a:1;   // since a is not adjacent to c.
    unsigned int b;
    unsigned int c:1;
    unsigned int d;
};
```

In that example, since `a` is not adjacent to `c`, they are both ``packed'' in their own `int`s.

So we have one `int` each for `a`, `b`, `c`, and `d`. Since my `int`s are 4 bytes, that's a grand total of 16 bytes.

A quick rearrangement yields some space savings from 16 bytes down to 12 bytes (on my system):

```
struct foo {            // sizeof(struct foo) == 12 (for me)
    unsigned int a:1;
    unsigned int c:1;
    unsigned int b;
    unsigned int d;
};
```

And now, since `a` is next to `c`, the compiler puts them together into a single `int`.

So we have one `int` for a combined `a` and `c`, and one `int` each for `b` and `d`. For a grand total of 3 `int`s, or 12 bytes.

Put all your bitfields together to get the compiler to combine them.

20.8.2 Signed or Unsigned `int`s

If you just declare a bit-field to be `int`, the different compilers will treat it as `signed` or `unsigned`. Just like the situation with `char`.

Be specific about the signedness when using bit-fields.

20.8.3 Unnamed Bit-Fields

In some specific circumstances, you might need to reserve some bits for hardware reasons, but not need to use them in code.

For example, let's say you have a byte where the top 2 bits have a meaning, the bottom 1 bit has a meaning, but the middle 5 bits do not get used by you[4].

We *could* do something like this:

```
struct foo {
    unsigned char a:2;
    unsigned char dummy:5;
    unsigned char b:1;
};
```

And that works---in our code we use `a` and `b`, but never `dummy`. It's just there to eat up 5 bits to make sure `a` and `b` are in the ``required'' (by this contrived example) positions within the byte.

C allows us a way to clean this up: *unnamed bit-fields*. You can just leave the name (`dummy`) out in this case, and C is perfectly happy for the same effect:

```
struct foo {
    unsigned char a:2;
    unsigned char :5;   // <-- unnamed bit-field!
    unsigned char b:1;
};
```

20.8.4 Zero-Width Unnamed Bit-Fields

Some more esoterica out here... Let's say you were packing bits into an `unsigned int`, and you needed some adjacent bit-fields to pack into the *next* `unsigned int`.

[4]Assuming 8-bit `char`s, i.e. `CHAR_BIT == 8`.

That is, if you do this:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int c:3;
    unsigned int d:4;
};
```

the compiler packs all those into a single `unsigned int`. But what if you needed `a` and `b` in one `int`, and `c` and `d` in a different one?

There's a solution for that: put an unnamed bit-field of width `0` where you want the compiler to start anew with packing bits in a different `int`:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int :0;   // <--Zero-width unnamed bit-field
    unsigned int c:3;
    unsigned int d:4;
};
```

It's analogous to an explicit page break in a word processor. You're telling the compiler, ``Stop packing bits in this `unsigned`, and start packing them in the next one."

By adding the zero-width unnamed bit field in that spot, the compiler puts `a` and `b` in one `unsigned int`, and `c` and `d` in another `unsigned int`. Two total, for a size of 8 bytes on my system (`unsigned ints` are 4 bytes each).

20.9 Unions

These are basically just like `structs`, except the fields overlap in memory. The `union` will be only large enough for the largest field, and you can only use one field at a time.

It's a way to reuse the same memory space for different types of data.

You declare them just like `structs`, except it's `union`. Take a look at this:

```
union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};
```

Now, that's a lot of fields. If this were a `struct`, my system would tell me it took 36 bytes to hold it all.

But it's a `union`, so all those fields overlap in the same stretch of memory. The biggest one is `int` (or `float`), taking up 4 bytes on my system. And, indeed, if I ask for the `sizeof` the `union foo`, it tells me 4!

The tradeoff is that you can only portably use one of those fields at a time. However…

20.9.1 Unions and Type Punning

You can non-portably write to one `union` field and read from another!

Doing so is called type punning[5], and you'd use it if you really knew what you were doing, typically with some kind of low-level programming.

[5] https://en.wikipedia.org/wiki/Type_punning

Since the members of a union share the same memory, writing to one member necessarily affects the others. And if you read from one what was written to another, you get some weird effects.

```
#include <stdio.h>

union foo {
    float b;
    short a;
};

int main(void)
{
    union foo x;

    x.b = 3.14159;

    printf("%f\n", x.b);  // 3.14159, fair enough

    printf("%d\n", x.a);  // But what about this?
}
```

On my system, this prints out:

```
3.141590
4048
```

because under the hood, the object representation for the float `3.14159` was the same as the object representation for the short `4048`. On my system. Your results may vary.

20.9.2 Pointers to `unions`

If you have a pointer to a `union`, you can cast that pointer to any of the types of the fields in that `union` and get the values out that way.

In this example, we see that the `union` has `int`s and `float`s in it. And we get pointers to the `union`, but we cast them to `int*` and `float*` types (the cast silences compiler warnings). And then if we dereference those, we see that they have the values we stored directly in the `union`.

```
#include <stdio.h>

union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};

int main(void)
{
    union foo x;

    int *foo_int_p = (int *)&x;
    float *foo_float_p = (float *)&x;

    x.a = 12;
    printf("%d\n", x.a);           // 12
    printf("%d\n", *foo_int_p);    // 12, again

```

```
20      x.g = 3.141592;
21      printf("%f\n", x.g);           // 3.141592
22      printf("%f\n", *foo_float_p);  // 3.141592, again
23  }
```

The reverse is also true. If we have a pointer to a type inside the `union`, we can cast that to a pointer to the `union` and access its members.

```
union foo x;
int *foo_int_p = (int *)&x;             // Pointer to int field
union foo *p = (union foo *)foo_int_p;  // Back to pointer to union

p->a = 12;  // This line the same as...
x.a = 12;   // this one.
```

All this just lets you know that, under the hood, all these values in a `union` start at the same place in memory, and that's the same as where the entire `union` is.

20.9.3 Common Initial Sequences in Unions

If you have a `union` of `structs`, and all those `structs` begin with a *common initial sequence*, it's valid to access members of that sequence from any of the `union` members.

What?

Here are two `structs` with a common initial sequence:

```
struct a {
    int x;     //
    float y;   // Common initial sequence

    char *p;
};

struct b {
    int x;     //
    float y;   // Common initial sequence

    double *p;
    short z;
};
```

Do you see it? It's that they start with `int` followed by `float`---that's the common initial sequence. The members in the sequence of the `structs` have to be compatible types. And we see that with `x` and `y`, which are `int` and `float` respectively.

Now let's build a union of these:

```
union foo {
    struct a sa;
    struct b sb;
};
```

What this rule tells us is that we're guaranteed that the members of the common initial sequences are interchangeable in code. That is:

- `f.sa.x` is the same as `f.sb.x`.

and

- `f.sa.y` is the same as `f.sb.y`.

Because fields `x` and `y` are both in the common initial sequence.

Also, the names of the members in the common initial sequence don't matter---all that matters is that the types are the same.

All together, this allows us a way to safely add some shared information between `structs` in the `union`. The best example of this is probably using a field to determine the type of `struct` out of all the `structs` in the `union` that is currently ``in use''.

That is, if we weren't allowed this and we passed the `union` to some function, how would that function know which member of the `union` was the one it should look at?

Take a look at these `structs`. Note the common initial sequence:

```
#include <stdio.h>

struct common {
    int type;   // common initial sequence
};

struct antelope {
    int type;   // common initial sequence

    int loudness;
};

struct octopus {
    int type;   // common initial sequence

    int sea_creature;
    float intelligence;
};
```

Now let's throw them into a `union`:

```
union animal {
    struct common common;
    struct antelope antelope;
    struct octopus octopus;
};
```

Also, please indulge me these two `#defines` for the demo:

```
#define ANTELOPE 1
#define OCTOPUS  2
```

So far, nothing special has happened here. It seems like the `type` field is completely useless.

But now let's make a generic function that prints a `union animal`. It has to somehow be able to tell if it's looking at a `struct antelope` or a `struct octopus`.

Because of the magic of common initial sequences, it can look up the animal type in any of these places for a particular `union animal x`:

```
int type = x.common.type;    \\ or...
int type = x.antelope.type;  \\ or...
int type = x.octopus.type;
```

All those refer to the same value in memory.

And, as you might have guessed, the `struct common` is there so code can agnostically look at the type without mentioning a particular animal.

Let's look at the code to print a `union animal`:

```
29 void print_animal(union animal *x)
30 {
31     switch (x->common.type) {
32         case ANTELOPE:
33             printf("Antelope: loudness=%d\n", x->antelope.loudness);
34             break;
35 
36         case OCTOPUS:
37             printf("Octopus : sea_creature=%d\n", x->octopus.sea_creature);
38             printf("          intelligence=%f\n", x->octopus.intelligence);
39             break;
40 
41         default:
42             printf("Unknown animal type\n");
43     }
44 
45 }
46 
47 int main(void)
48 {
49     union animal a = {.antelope.type=ANTELOPE, .antelope.loudness=12};
50     union animal b = {.octopus.type=OCTOPUS, .octopus.sea_creature=1,
51                                       .octopus.intelligence=12.8};
52 
53     print_animal(&a);
54     print_animal(&b);
55 }
```

See how on line 29 we're just passing in the `union`---we have no idea what type of animal `struct` is in use within it.

But that's OK! Because on line 31 we check the type to see if it's an antelope or an octopus. And then we can look at the proper `struct` to get the members.

It's definitely possible to get this same effect using just `structs`, but you can do it this way if you want the memory-saving effects of a `union`.

20.10 Unions and Unnamed Structs

You know how you can have an unnamed `struct`, like this:

```
struct {
    int x, y;
} s;
```

That defines a variable `s` that is of anonymous `struct` type (because the `struct` has no name tag), with members `x` and `y`.

So things like this are valid:

```
s.x = 34;
s.y = 90;
```

```
printf("%d %d\n", s.x, s.y);
```

Turns out you can drop those unnamed `structs` in `unions` just like you might expect:

```
union foo {
    struct {        // unnamed!
        int x, y;
    } a;

    struct {        // unnamed!
        int z, w;
    } b;
};
```

And then access them as per normal:

```
union foo f;

f.a.x = 1;
f.a.y = 2;
f.b.z = 3;
f.b.w = 4;
```

No problem!

20.11 Passing and Returning `structs` and `unions`

You can pass a `struct` or `union` to a function by value (as opposed to a pointer to it)---a copy of that object to the parameter will be made as if by assignment as per usual.

You can also return a `struct` or `union` from a function and it is also passed by value back.

```
#include <stdio.h>

struct foo {
    int x, y;
};

struct foo f(void)
{
    return (struct foo){.x=34, .y=90};
}

int main(void)
{
    struct foo a = f();  // Copy is made

    printf("%d %d\n", a.x, a.y);
}
```

Fun fact: if you do this, you can use the `.` operator right off the function call:

```
    printf("%d %d\n", f().x, f().y);
```

(Of course that example calls the function twice, inefficiently.)

And the same holds true for returning pointers to `structs` and `unions`---just be sure to use the `->` arrow operator in that case.

Chapter 21

Characters and Strings II

We've talked about how `char` types are actually just small integer types… but it's the same for a character in single quotes.

But a string in double quotes is type `const char *`.

Turns out there are few more types of strings and characters, and it leads down one of the most infamous rabbit holes in the language: the whole multibyte/wide/Unicode/localization thingy.

We're going to peer into that rabbit hole, but not go in. …Yet!

21.1 Escape Sequences

We're used to strings and characters with regular letters, punctuation, and numbers:

```
char *s = "Hello!";
char t = 'c';
```

But what if we want some special characters in there that we can't type on the keyboard because they don't exist (e.g. ``€''), or even if we want a character that's a single quote? We clearly can't do this:

```
char t = ''';
```

To do these things, we use something called *escape sequences*. These are the backslash character (\) followed by another character. The two (or more) characters together have special meaning.

For our single quote character example, we can put an escape (that is, \) in front of the central single quote to solve it:

```
char t = '\'';
```

Now C knows that `\'` means just a regular quote we want to print, not the end of the character sequence.

You can say either ``backslash'' or ``escape'' in this context (``escape that quote'') and C devs will know what you're talking about. Also, ``escape'' in this context is different than your `Esc` key or the ASCII `ESC` code.

21.1.1 Frequently-used Escapes

In my humble opinion, these escape characters make up 99.2%[1] of all escapes.

[1] I just made up that number, but it's probably not far off

Code	Description
`\n`	Newline character---when printing, continue subsequent output on the next line
`\'`	Single quote---used for a single quote character constant
`\"`	Double quote---used for a double quote in a string literal
`\\`	Backslash---used for a literal `\` in a string or character

Here are some examples of the escapes and what they output when printed.

```
printf("Use \\n for newline\n");  // Use \n for newline
printf("Say \"hello\"!\n");       // Say "hello"!
printf("%c\n", '\'');             // '
```

21.1.2 Rarely-used Escapes

But there are more escapes! You just don't see these as often.

Code	Description
`\a`	Alert. This makes the terminal make a sound or flash, or both!
`\b`	Backspace. Moves the cursor back a character. Doesn't delete the character.
`\f`	Formfeed. This moves to the next ``page", but that doesn't have much modern meaning. On my system, this behaves like `\v`.
`\r`	Return. Move to the beginning of the same line.
`\t`	Horizontal tab. Moves to the next horizontal tab stop. On my machine, this lines up on columns that are multiples of 8, but YMMV.
`\v`	Vertical tab. Moves to the next vertical tab stop. On my machine, this moves to the same column on the next line.
`\?`	Literal question mark. Sometimes you need this to avoid trigraphs, as shown below.

21.1.2.1 Single Line Status Updates

A use case for `\b` or `\r` is to show status updates that appear on the same line on the screen and don't cause the display to scroll. Here's an example that does a countdown from 10. (Note this makes use of the non-standard POSIX function `sleep()` from `<unistd.h>`---if you're not on a Unix-like, search for your platform and `sleep` for the equivalent.)

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

int main(void)
{
    for (int i = 10; i >= 0; i--) {
        printf("\rT minus %d second%s... \b", i, i != 1? "s": "");

        fflush(stdout);  // Force output to update

        // Sleep for 1 second
        thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
    }

    printf("\rLiftoff!             \n");
}
```

Quite a few things are happening on line 7. First of all, we lead with a `\r` to get us to the beginning of the current line, then we overwrite whatever's there with the current countdown. (There's ternary operator out there to make sure we print `1 second` instead of `1 seconds`.)

Also, there's a space after the `...` That's so that we properly overwrite the last `.` when `i` drops from `10` to `9` and we get a column narrower. Try it without the space to see what I mean.

And we wrap it up with a `\b` to back up over that space so the cursor sits at the exact end of the line in an aesthetically-pleasing way.

Note that line 14 also has a lot of spaces at the end to overwrite the characters that were already there from the countdown.

Finally, we have a weird `fflush(stdout)` in there, whatever that means. Short answer is that most terminals are *line buffered* by default, meaning they don't actually display anything until a newline character is encountered. Since we don't have a newline (we just have `\r`), without this line, the program would just sit there until `Liftoff!` and then print everything all in one instant. `fflush()` overrides this behavior and forces output to happen *right now*.

21.1.2.2 The Question Mark Escape

Why bother with this? After all, this works just fine:

```
printf("Doesn't it?\n");
```

And it works fine with the escape, too:

```
printf("Doesn't it\?\n");   // Note \?
```

So what's the point??!

Let's get more emphatic with another question mark and an exclamation point:

```
printf("Doesn't it??!\n");
```

When I compile this, I get this warning:

```
foo.c: In function ‘main’:
foo.c:5:23: warning: trigraph ??! converted to | [-Wtrigraphs]
    5 |     printf("Doesn't it??!\n");
      |
```

And running it gives this unlikely result:

```
Doesn't it|
```

So *trigraphs*? What the heck is this??!

I'm sure we'll revisit this dusty corner of the language later, but the short of it is the compiler looks for certain triplets of characters starting with `??` and it substitutes other characters in their place. So if you're on some ancient terminal without a pipe symbol (`|`) on the keyboard, you can type `??!` instead.

You can fix this by escaping the second question mark, like so:

```
printf("Doesn't it?\?!\n");
```

And then it compiles and works as-expected.

These days, of course, no one ever uses trigraphs. But that whole `??!` does sometimes appear if you decide to use it in a string for emphasis.

21.1.3 Numeric Escapes

In addition, there are ways to specify numeric constants or other character values inside strings or character constants.

If you know an octal or hexadecimal representation of a byte, you can include that in a string or character constant.

The following table has example numbers, but any hex or octal numbers may be used. Pad with leading zeros if necessary to read the proper digit count.

Code	Description
`\123`	Embed the byte with octal value `123`, 3 digits exactly.
`\x4D`	Embed the byte with hex value `4D`, 2 digits.
`\u2620`	Embed the Unicode character at code point with hex value `2620`, 4 digits.
`\U0001243F`	Embed the Unicode character at code point with hex value `1243F`, 8 digits.

Here's an example of the less-commonly used octal notation to represent the letter `B` in between `A` and `C`. Normally this would be used for some kind of special unprintable character, but we have other ways to do that, below, and this is just an octal demo:

```
printf("A\102C\n");  // 102 is `B` in ASCII/UTF-8
```

Note there's no leading zero on the octal number when you include it this way. But it does need to be three characters, so pad with leading zeros if you need to.

But far more common is to use hex constants these days. Here's a demo that you shouldn't use, but it demos embedding the UTF-8 bytes 0xE2, 0x80, and 0xA2 in a string, which corresponds to the Unicode ``bullet'' character (•).

```
printf("\xE2\x80\xA2 Bullet 1\n");
printf("\xE2\x80\xA2 Bullet 2\n");
printf("\xE2\x80\xA2 Bullet 3\n");
```

Produces the following output if you're on a UTF-8 console (or probably garbage if you're not):

```
• Bullet 1
• Bullet 2
• Bullet 3
```

But that's a crummy way to do Unicode. You can use the escapes `\u` (16-bit) or `\U` (32-bit) to just refer to Unicode by code point number. The bullet is `2022` (hex) in Unicode, so you can do this and get more portable results:

```
printf("\u2022 Bullet 1\n");
printf("\u2022 Bullet 2\n");
printf("\u2022 Bullet 3\n");
```

Be sure to pad `\u` with enough leading zeros to get to four characters, and `\U` with enough to get to eight.

For example, that bullet could be done with `\U` and four leading zeros:

```
printf("\U00002022 Bullet 1\n");
```

But who has time to be that verbose?

Chapter 22

Enumerated Types: `enum`

C offers us another way to have constant integer values by name: `enum`.

For example:

```
enum {
  ONE=1,
  TWO=2
};

printf("%d %d", ONE, TWO);  // 1 2
```

In some ways, it can be better---or different---than using a `#define`. Key differences:

- `enum`s can only be integer types.
- `#define` can define anything at all.
- `enum`s are often shown by their symbolic identifier name in a debugger.
- `#define`d numbers just show as raw numbers which are harder to know the meaning of while debugging.

Since they're integer types, they can be used any place integers can be used, including in array dimensions and `case` statements.

Let's tear into this more.

22.1 Behavior of `enum`

22.1.1 Numbering

`enum`s are automatically numbered unless you override them.

They start at `0`, and autoincrement up from there, by default:

```
enum {
    SHEEP,  // Value is 0
    WHEAT,  // Value is 1
    WOOD,   // Value is 2
    BRICK,  // Value is 3
    ORE     // Value is 4
};

printf("%d %d\n", SHEEP, BRICK);  // 0 3
```

You can force particular integer values, as we saw earlier:

```
enum {
  X=2,
  Y=18,
  Z=-2
};
```

Duplicates are not a problem:

```
enum {
  X=2,
  Y=2,
  Z=2
};
```

if values are omitted, numbering continues counting in the positive direction from whichever value was last specified. For example:

```
enum {
  A,    // 0, default starting value
  B,    // 1
  C=4,  // 4, manually set
  D,    // 5
  E,    // 6
  F=3   // 3, manually set
  G,    // 4
  H     // 5
}
```

22.1.2 Trailing Commas

This is perfectly fine, if that's your style:

```
enum {
  X=2,
  Y=18,
  Z=-2,   // <-- Trailing comma
};
```

It's gotten more popular in languages of the recent decades so you might be pleased to see it.

22.1.3 Scope

`enum`s scope as you'd expect. If at file scope, the whole file can see it. If in a block, it's local to that block.

It's really common for `enum`s to be defined in header files so they can be `#included` at file scope.

22.1.4 Style

As you've noticed, it's common to declare the `enum` symbols in uppercase (with underscores).

This isn't a requirement, but is a very, very common idiom.

22.2 Your `enum` is a Type

This is an important thing to know about `enum`: they're a type, analogous to how a `struct` is a type.

You can give them a tag name so you can refer to the type later and declare variables of that type.

Now, since `enum`s are integer types, why not just use `int`?

In C, the best reason for this is code clarity--it's a nice, typed way to describe your thinking in code. C (unlike C++) doesn't actually enforce any values being in range for a particular `enum`.

Let's do an example where we declare a variable `r` of type `enum resource` that can hold those values:

```
// Named enum, type is "enum resource"

enum resource {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
};

// Declare a variable "r" of type "enum resource"

enum resource r = BRICK;

if (r == BRICK) {
    printf("I'll trade you a brick for two sheep.\n");
}
```

You can also `typedef` these, of course, though I personally don't like to.

```
typedef enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} RESOURCE;

RESOURCE r = BRICK;
```

Another shortcut that's legal but rare is to declare variables when you declare the `enum`:

```
// Declare an enum and some initialized variables of that type:

enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

You can also give the `enum` a name so you can use it later, which is probably what you want to do in most cases:

```
// Declare an enum and some initialized variables of that type:

enum resource {   // <-- type is now "enum resource"
    SHEEP,
```

```
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

In short, enums are a great way to write nice, scoped, typed, clean code.

Chapter 23

Pointers III: Pointers to Pointers and More

Here's where we cover some intermediate and advanced pointer usage. If you don't have pointers down well, review the previous chapters on pointers and pointer arithmetic before starting on this stuff.

23.1 Pointers to Pointers

If you can have a pointer to a variable, and a variable can be a pointer, can you have a pointer to a variable that it itself a pointer?

Yes! This is a pointer to a pointer, and it's held in variable of type pointer-pointer.

Before we tear into that, I want to try for a *gut feel* for how pointers to pointers work.

Remember that a pointer is just a number. It's a number that represents an index in computer memory, typically one that holds a value we're interested in for some reason.

That pointer, which is a number, has to be stored somewhere. And that place is memory, just like everything else[1].

But because it's stored in memory, it must have an index it's stored at, right? The pointer must have an index in memory where it is stored. And that index is a number. It's the address of the pointer. It's a pointer to the pointer.

Let's start with a regular pointer to an `int`, back from the earlier chapters:

```
#include <stdio.h>

int main(void)
{
    int x = 3490;  // Type: int
    int *p = &x;   // Type: pointer to an int

    printf("%d\n", *p);  // 3490
}
```

[1]There's some devil in the details with values that are stored in registers only, but we can safely ignore that for our purposes here. Also the C spec makes no stance on these ``register'' things beyond the `register` keyword, the description for which doesn't mention registers.

Straightforward enough, right? We have two types represented: `int` and `int*`, and we set up `p` to point to `x`. Then we can dereference `p` on line 8 and print out the value `3490`.

But, like we said, we can have a pointer to any variable… so does that mean we can have a pointer to `p`?

In other words, what type is this expression?

```
int x = 3490;  // Type: int
int *p = &x;   // Type: pointer to an int

&p  // <-- What type is the address of p? AKA a pointer to p?
```

If `x` is an `int`, then `&x` is a pointer to an `int` that we've stored in `p` which is type `int*`. Follow? (Repeat this paragraph until you do!)

And therefore `&p` is a pointer to an `int*`, AKA a ``pointer to a pointer to an `int`". AKA ```int`-pointer-pointer".

Got it? (Repeat the previous paragraph until you do!)

We write this type with two asterisks: `int **`. Let's see it in action.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 3490;  // Type: int
6      int *p = &x;   // Type: pointer to an int
7      int **q = &p;  // Type: pointer to pointer to int
8
9      printf("%d %d\n", *p, **q);  // 3490 3490
10 }
```

Let's make up some pretend addresses for the above values as examples and see what these three variables might look like in memory. The address values, below are just made up by me for example purposes:

Variable	Stored at Address	Value Stored There
`x`	`28350`	`3490`---the value from the code
`p`	`29122`	`28350`---the address of `x`!
`q`	`30840`	`29122`---the address of `p`!

Indeed, let's try it for real on my computer[2] and print out the pointer values with `%p` and I'll do the same table again with actual references (printed in hex).

Variable	Stored at Address	Value Stored There
`x`	`0x7ffd96a07b94`	`3490`---the value from the code
`p`	`0x7ffd96a07b98`	`0x7ffd96a07b94`---the address of `x`!
`q`	`0x7ffd96a07ba0`	`0x7ffd96a07b98`---the address of `p`!

You can see those addresses are the same except the last byte, so just focus on those.

On my system, `int`s are 4 bytes, which is why we're seeing the address go up by 4 from `x` to `p`[3] and then goes up by 8 from `p` to `q`. On my system, all pointers are 8 bytes.

[2] You're very likely to get different numbers on yours.

[3] There is absolutely nothing in the spec that says this will always work this way, but it happens to work this way on my system.

Does it matter if it's an `int*` or an `int**`? Is one more bytes than the other? Nope! Remember that all pointers are addresses, that is indexes into memory. And on my machine you can represent an index with 8 bytes… doesn't matter what's stored at that index.

Now check out what we did there on line 9 of the previous example: we *double dereferenced* `q` to get back to our `3490`.

This is the important bit about pointers and pointers to pointers:

- You can get a pointer to anything with `&` (including to a pointer!)
- You can get the thing a pointer points to with `*` (including a pointer!)

So you can think of `&` as being used to make pointers, and `*` being the inverse---it goes the opposite direction of `&`---to get to the thing pointed to.

In terms of type, each time you `&`, that adds another pointer level to the type.

If you have	Then you run	The result type is
`int x`	`&x`	`int *`
`int *x`	`&x`	`int **`
`int **x`	`&x`	`int ***`
`int ***x`	`&x`	`int ****`

And each time you use dereference (`*`), it does the opposite:

If you have	Then you run	The result type is
`int ****x`	`*x`	`int ***`
`int ***x`	`*x`	`int **`
`int **x`	`*x`	`int *`
`int *x`	`*x`	`int`

Note that you can use multiple `*`s in a row to quickly dereference, just like we saw in the example code with `**q`, above. Each one strips away one level of indirection.

If you have	Then you run	The result type is
`int ****x`	`***x`	`int *`
`int ***x`	`**x`	`int *`
`int **x`	`**x`	`int`

In general, `&*E == E`[4]. The dereference ``undoes" the address-of.

But `&` doesn't work the same way---you can only do those one at a time, and have to store the result in an intermediate variable:

```
int x = 3490;     // Type: int
int *p = &x;      // Type: int *, pointer to an int
int **q = &p;     // Type: int **, pointer to pointer to int
int ***r = &q;    // Type: int ***, pointer to pointer to pointer to int
int ****s = &r;   // Type: int ****, you get the idea
int *****t = &s;  // Type: int *****
```

[4]Even if `E` is `NULL`, it turns out, weirdly.

23.1.1 Pointer Pointers and `const`

If you recall, declaring a pointer like this:

```
int *const p;
```

means that you can't modify `p`. Trying to `p++` would give you a compile-time error.

But how does that work with `int **` or `int ***`? Where does the `const` go, and what does it mean?

Let's start with the simple bit. The `const` right next to the variable name refers to that variable. So if you want an `int***` that you can't change, you can do this:

```
int ***const p;

p++;  // Not allowed
```

But here's where things get a little weird.

What if we had this situation:

```
1 int main(void)
2 {
3     int x = 3490;
4     int *const p = &x;
5     int **q = &p;
6 }
```

When I build that, I get a warning:

```
warning: initialization discards ‘const’ qualifier from pointer target type
    7 |     int **q = &p;
      |               ^
```

What's going on? The compiler is telling us here that we had a variable that was `const`, and we're assigning its value into another variable that is not `const` in the same way. The ``constness'' is discarded, which probably isn't what we wanted to do.

The type of `p` is `int *const p`, and so `&p` is type `int *const *`. And we try to assign that into `q`.

But `q` is `int **`! A type with different `const`ness on the first `*`! So we get a warning that the `const` in `p`'s `int *const *` is being ignored and thrown away.

We can fix that by making sure `q`'s type is at least as `const` as `p`.

```
int x = 3490;
int *const p = &x;
int *const *q = &p;
```

And now we're happy.

We could make `q` even more `const`. As it is, above, we're saying, ```q` isn't itself `const`, but the thing it points to is `const`.'' But we could make them both `const`:

```
int x = 3490;
int *const p = &x;
int *const *const q = &p;  // More const!
```

And that works, too. Now we can't modify `q`, or the pointer `q` points to.

23.2 Multibyte Values

We kinda hinted at this in a variety of places earlier, but clearly not every value can be stored in a single byte of memory. Things take up multiple bytes of memory (assuming they're not `chars`). You can tell how many bytes by using `sizeof`. And you can tell which address in memory is the *first* byte of the object by using the standard `&` operator, which always returns the address of the first byte.

And here's another fun fact! If you iterate over the bytes of any object, you get its *object representation*. Two things with the same object representation in memory are equal.

If you want to iterate over the object representation, you should do it with pointers to `unsigned char`.

Let's make our own version of `memcpy()`[5] that does exactly this:

```
void *my_memcpy(void *dest, const void *src, size_t n)
{
    // Make local variables for src and dest, but of type unsigned char

    const unsigned char *s = src;
    unsigned char *d = dest;

    while (n-- > 0)   // For the given number of bytes
        *d++ = *s++;  // Copy source byte to dest byte

    // Most copy functions return a pointer to the dest as a convenience
    // to the caller

    return dest;
}
```

(There are some good examples of post-increment and post-decrement in there for you to study, as well.)

It's important to note that the version, above, is probably less efficient than the one that comes with your system.

But you can pass pointers to anything into it, and it'll copy those objects. Could be `int*`, `struct animal*`, or anything.

Let's do another example that prints out the object representation bytes of a `struct` so we can see if there's any padding in there and what values it has[6].

```
#include <stdio.h>

struct foo {
    char a;
    int b;
};

int main(void)
{
    struct foo x = {0x12, 0x12345678};
    unsigned char *p = (unsigned char *)&x;

    for (size_t i = 0; i < sizeof x; i++) {
        printf("%02X\n", p[i]);
```

[5] https://beej.us/guide/bgclr/html/split/stringref.html#man-memcpy

[6] Your C compiler is not required to add padding bytes, and the values of any padding bytes that are added are indeterminate.

```
15      }
16  }
```

What we have there is a `struct foo` that's built in such a way that should encourage a compiler to inject padding bytes (though it doesn't have to). And then we get an `unsigned char *` to the first byte of the `struct foo` variable `x`.

From there, all we need to know is the `sizeof x` and we can loop through that many bytes, printing out the values (in hex for ease).

Running this gives a bunch of numbers as output. I've annotated it below to identify where the values were stored:

```
12  | x.a == 0x12

AB  |
BF  | padding bytes with "random" value
26  |

78  |
56  | x.b == 0x12345678
34  |
12  |
```

On all systems, `sizeof(char)` is 1, and we see that first byte at the top of the output holding the value `0x12` that we stored there.

Then we have some padding bytes---for me, these varied from run to run.

Finally, on my system, `sizeof(int)` is 4, and we can see those 4 bytes at the end. Notice how they're the same bytes as are in the hex value `0x12345678`, but strangely in reverse order[7].

So that's a little peek under the hood at the bytes of a more complex entity in memory.

23.3 The `NULL` Pointer and Zero

These things can be used interchangeably:

- `NULL`
- `0`
- `'\0'`
- `(void *)0`

Personally, I always use `NULL` when I mean `NULL`, but you might see some other variants from time to time. Though `'\0'` (a byte with all bits set to zero) will also compare equal, it's *weird* to compare it to a pointer; you should compare `NULL` against the pointer. (Of course, lots of times in string processing, you're comparing *the thing the pointer points to* to `'\0'`, and that's right.)

`0` is called the *null pointer constant*, and, when compared to or assigned into another pointer, it is converted to a null pointer of the same type.

23.4 Pointers as Integers

You can cast pointers to integers and vice-versa (since a pointer is just an index into memory), but you probably only ever need to do this if you're doing some low-level hardware stuff. The results of such machinations

[7] This will vary depending on the architecture, but my system is *little endian*, which means the least-significant byte of the number is stored first. *Big endian* systems will have the `12` first and the `78` last. But the spec doesn't dictate anything about this representation.

are implementation-defined, so they aren't portable. And *weird things* could happen.

C does make one guarantee, though: you can convert a pointer to a `uintptr_t` type and you'll be able to convert it back to a pointer without losing any data.

`uintptr_t` is defined in `<stdint.h>`[8].

Additionally, if you feel like being signed, you can use `intptr_t` to the same effect.

23.5 Casting Pointers to other Pointers

There's only one safe pointer conversion:

1. Converting to `intptr_t` or `uintptr_t`.
2. Converting to and from `void*`.

TWO! Two safe pointer conversions.

3. Converting to and from `char*` (or `signed char*`/`unsigned char*`).

THREE! Three safe conversions!

4. Converting to and from a pointer to a `struct` and a pointer to its first member, and vice-versa.

FOUR! Four safe conversions!

If you cast to a pointer of another type and then access the object it points to, the behavior is undefined due to something called *strict aliasing*.

Plain old *aliasing* refers to the ability to have more than one way to access the same object. The access points are aliases for each other.

Strict aliasing says you are only allowed to access an object via pointers to *compatible types* to that object.

For example, this is definitely allowed:

```
int a = 1;
int *p = &a;
```

`p` is a pointer to an `int`, and it points to a compatible type---namely `int`---so we're golden.

But the following isn't good because `int` and `float` are not compatible types:

```
int a = 1;
float *p = (float *)&a;
```

Here's a demo program that does some aliasing. It takes a variable `v` of type `int32_t` and aliases it to a pointer to a `struct words`. That `struct` has two `int16_t`s in it. These types are incompatible, so we're in violation of strict aliasing rules. The compiler will assume that these two pointers never point to the same object… but we're making it so they do. Which is naughty of us.

Let's see if we can break something.

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

struct words {
    int16_t v[2];
};

void fun(int32_t *pv, struct words *pw)
```

[8] It's an optional feature, so it might not be there---but it probably is.

```
{
    for (int i = 0; i < 5; i++) {
        (*pv)++;

        // Print the 32-bit value and the 16-bit values:

        printf("%x, %x-%x\n", *pv, pw->v[1], pw->v[0]);
    }
}

int main(void)
{
    int32_t v = 0x12345678;

    struct words *pw = (struct words *)&v;  // Violates strict aliasing

    fun(&v, pw);
}
```

See how I pass in the two incompatible pointers to `fun()`? One of the types is `int32_t*` and the other is `struct words*`.

But they both point to the same object: the 32-bit value initialized to `0x12345678`.

So if we look at the fields in the `struct words`, we should see the two 16-bit halves of that number. Right?

And in the `fun()` loop, we increment the pointer to the `int32_t`. That's it. But since the `struct` points to that same memory, it, too, should be updated to the same value.

So let's run it and get this, with the 32-bit value on the left and the two 16-bit portions on the right. It should match[9]:

```
12345679, 1234-5679
1234567a, 1234-567a
1234567b, 1234-567b
1234567c, 1234-567c
1234567d, 1234-567d
```

and it does… *UNTIL TOMORROW!*

Let's try it compiling GCC with `-O3` and `-fstrict-aliasing`:

```
12345679, 1234-5678
1234567a, 1234-5679
1234567b, 1234-567a
1234567c, 1234-567b
1234567d, 1234-567c
```

They're off by one! But they point to the same memory! How could this be? Answer: it's undefined behavior to alias memory like that. *Anything is possible*, except not in a good way.

If your code violates strict aliasing rules, whether it works or not depends on how someone decides to compile it. And that's a bummer since that's beyond your control. Unless you're some kind of omnipotent deity.

Unlikely, sorry.

GCC can be forced to not use the strict aliasing rules with `-fno-strict-aliasing`. Compiling the demo program, above, with `-O3` and this flag causes the output to be as expected.

[9] I'm printing out the 16-bit values reversed since I'm on a little-endian machine and it makes it easier to read here.

Lastly, *type punning* is using pointers of different types to look at the same data. Before strict aliasing, this kind of things was fairly common:

```
int a = 0x12345678;
short b = *((short *)&a);   // Violates strict aliasing
```

If you want to do type punning (relatively) safely, see the section on Unions and Type Punning.

23.6 Pointer Differences

As you know from the section on pointer arithmetic, you can subtract one pointer from another[10] to get the difference between them in count of array elements.

Now the *type of that difference* is something that's up to the implementation, so it could vary from system to system.

To be more portable, you can store the result in a variable of type `ptrdiff_t` defined in `<stddef.h>`.

```
int cats[100];

int *f = cats + 20;
int *g = cats + 60;

ptrdiff_t d = g - f;  // difference is 40
```

And you can print it by prefixing the integer format specifier with `t`:

```
printf("%td\n", d);  // Print decimal: 40
printf("%tX\n", d);  // Print hex:     28
```

23.7 Pointers to Functions

Functions are just collections of machine instructions in memory, so there's no reason we can't get a pointer to the first instruction of the function.

And then call it.

This can be useful for passing a pointer to a function into another function as an argument. Then the second one could call whatever was passed in.

The tricky part with these, though, is that C needs to know the type of the variable that is the pointer to the function.

And it would really like to know all the details.

Like ``this is a pointer to a function that takes two `int` arguments and returns `void`".

How do you write all that down so you can declare a variable?

Well, it turns out it looks very much like a function prototype, except with some extra parentheses:

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int, int);
```

Also notice that you don't have to give the parameters names. But you can if you want; they're just ignored.

[10] Assuming they point to the same array object.

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int a, int b);
```

So now that we know how to declare a variable, how do we know what to assign into it? How do we get the address of a function?

Turns out there's a shortcut just like with getting a pointer to an array: you can just refer to the bare function name without parens. (You can put an `&` in front of this if you like, but it's unnecessary and not idiomatic.)

Once you have a pointer to a function, you can call it just by adding parens and an argument list.

Let's do a simple example where I effectively make an alias for a function by setting a pointer to it. Then we'll call it.

This code prints out `3490`:

```
#include <stdio.h>

void print_int(int n)
{
    printf("%d\n", n);
}

int main(void)
{
    // Assign p to point to print_int:

    void (*p)(int) = print_int;

    p(3490);          // Call print_int via the pointer
}
```

Notice how the type of `p` represents the return value and parameter types of `print_int`. It has to, or else C will complain about incompatible pointer types.

One more example here shows how we might pass a pointer to a function as an argument to another function.

We'll write a function that takes a couple integer arguments, plus a pointer to a function that operates on those two arguments. Then it prints the result.

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int mult(int a, int b)
{
    return a * b;
}

void print_math(int (*op)(int, int), int x, int y)
{
    int result = op(x, y);

```

```
17      printf("%d\n", result);
18  }
19
20  int main(void)
21  {
22      print_math(add, 5, 7);   // 12
23      print_math(mult, 5, 7);  // 35
24  }
```

Take a moment to digest that. The idea here is that we're going to pass a pointer to a function to `print_math()`, and it's going to call that function to do some math.

This way we can change the behavior of `print_math()` by passing another function into it. You can see we do that on lines 22-23 when we pass in pointers to functions `add` and `mult`, respectively.

Now, on line 13, I think we can all agree the function signature of `print_math()` is a sight to behold. And, if you can believe it, this one is actually pretty straight-forward compared to some things you can construct[11].

But let's digest it. Turns out there are only three parameters, but they're a little hard to see:

```
//                      op              x      y
//              |-----------------|  |---|  |---|
void print_math(int (*op)(int, int), int x, int y)
```

The first, `op`, is a pointer to a function that takes two `int`s as arguments and returns an `int`. This matches the signatures for both `add()` and `mult()`.

The second and third, `x` and `y`, are just standard `int` parameters.

Slowly and deliberately let your eyes play over the signature while you identify the working parts. One thing that always stands out for me is the sequence `(*op)(`, the parens and the asterisk. That's the giveaway it's a pointer to a function.

Finally, jump back to the *Pointers II* chapter for a pointer-to-function example using the built-in `qsort()`.

[11] The Go Programming Language drew its type declaration syntax inspiration from the opposite of what C does.

Chapter 24

Bitwise Operations

These numeric operations effectively allow you to manipulate individual bits in variables, fitting since C is such a low-level langauge[1].

If you're not familiar with bitwise operations, Wikipedia has a good bitwise article[2].

24.1 Bitwise AND, OR, XOR, and NOT

For each of these, the usual arithmetic conversions take place on the operands (which in this case must be an integer type), and then the appropriate bitwise operation is performed.

Operation	Operator	Example
AND	`&`	`a = b & c`
OR	`|`	`a = b | c`
XOR	`^`	`a = b ^ c`
NOT	`~`	`a = ~c`

Note how they're similar to the Boolean operators `&&` and `||`.

These have assignment shorthand variants similar to `+=` and `-=`:

Operator	Example	Longhand equivalent
`&=`	`a &= c`	`a = a & c`
`|=`	`a |= c`	`a = a | c`
`^=`	`a ^= c`	`a = a ^ c`

24.2 Bitwise Shift

For these, the integer promotions are performed on each operand (which must be an integer type) and then a bitwise shift is executed. The type of the result is the type of the promoted left operand.

New bits are filled with zeros, with a possible exception noted in the implementation-defined behavior, below.

[1] Not that other languages don't do this---they do. It is interesting how many modern languages use the same operators for bitwise that C does.

[2] https://en.wikipedia.org/wiki/Bitwise_operation

Operation	Operator	Example
Shift left	`<<`	`a = b << c`
Shift right	`>>`	`a = b >> c`

There's also the same similar shorthand for shifting:

Operator	Example	Longhand equivalent
`>>=`	`a >>= c`	`a = a >> c`
`<<=`	`a <<= c`	`a = a << c`

Watch for undefined behavior: no negative shifts, and no shifts that are larger than the size of the promoted left operand.

Also watch for implementation-defined behavior: if you right-shift a negative number, the results are implementation-defined. (It's perfectly fine to right-shift a signed `int`, just make sure it's positive.)

Chapter 25

Variadic Functions

Variadic is a fancy word for functions that take arbitrary numbers of arguments.

A regular function takes a specific number of arguments, for example:

```
int add(int x, int y)
{
    return x + y;
}
```

You can only call that with exactly two arguments which correspond to parameters `x` and `y`.

```
add(2, 3);
add(5, 12);
```

But if you try it with more, the compiler won't let you:

```
add(2, 3, 4);  // ERROR
add(5);        // ERROR
```

Variadic functions get around this limitation to a certain extent.

We've already seen a famous example in `printf()`! You can pass all kinds of things to it.

```
printf("Hello, world!\n");
printf("The number is %d\n", 2);
printf("The number is %d and pi is %f\n", 2, 3.14159);
```

It seems to not care how many arguments you give it!

Well, that's not entirely true. Zero arguments will give you an error:

```
printf();  // ERROR
```

This leads us to one of the limitations of variadic functions in C: they must have at least one argument.

But aside from that, they're pretty flexible, even allows arguments to have different types just like `printf()` does.

Let's see how they work!

25.1 Ellipses in Function Signatures

So how does it work, syntactically?

What you do is put all the arguments that *must* be passed first (and remember there has to be at least one) and after that, you put `...`. Just like this:

```
void func(int a, ...)   // Literally 3 dots here
```

Here's some code to demo that:

```
#include <stdio.h>

void func(int a, ...)
{
    printf("a is %d\n", a);  // Prints "a is 2"
}

int main(void)
{
    func(2, 3, 4, 5, 6);
}
```

So, great, we can get that first argument that's in variable `a`, but what about the rest of the arguments? How do you get to them?

Here's where the fun begins!

25.2 Getting the Extra Arguments

You're going to want to include `<stdarg.h>` to make any of this work.

First things first, we're going to use a special variable of type `va_list` (variable argument list) to keep track of which variable we're accessing at a time.

The idea is that we first start processing arguments with a call to `va_start()`, process each argument in turn with `va_arg()`, and then, when done, wrap it up with `va_end()`.

When you call `va_start()`, you need to pass in the *last named parameter* (the one just before the `...`) so it knows where to start looking for the additional arguments.

And when you call `va_arg()` to get the next argument, you have to tell it the type of argument to get next.

Here's a demo that adds together an arbitrary number of integers. The first argument is the number of integers to add together. We'll make use of that to figure out how many times we have to call `va_arg()`.

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

int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Start with arguments after "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Get the next int

        total += n;
    }

```

```
17      va_end(va);  // All done
18
19      return total;
20  }
21
22  int main(void)
23  {
24      printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
25      printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
26  }
```

(Note that when `printf()` is called, it uses the number of `%d`s (or whatever) in the format string to know how many more arguments there are!)

If the syntax of `va_arg()` is looking strange to you (because of that loose type name floating around in there), you're not alone. These are implemented with preprocessor macros in order to get all the proper magic in there.

25.3 `va_list` Functionality

What is that `va_list` variable we're using up there? It's an opaque variable[1] that holds information about which argument we're going to get next with `va_arg()`. You see how we just call `va_arg()` over and over? The `va_list` variable is a placeholder that's keeping track of progress so far.

But we have to initialize that variable to some sensible value. That's where `va_start()` comes into play.

When we called `va_start(va, count)`, above, we were saying, ``Initialize the `va` variable to point to the variable argument *immediately after* `count`."

And that's *why* we need to have at least one named variable in our argument list[2].

Once you have that pointer to the initial parameter, you can easily get subsequent argument values by calling `va_arg()` repeatedly. When you do, you have to pass in your `va_list` variable (so it can keep on keeping track of where you are), as well as the type of argument you're about to copy off.

It's up to you as a programmer to figure out which type you're going to pass to `va_arg()`. In the above example, we just did `int`s. But in the case of `printf()`, it uses the format specifier to determine which type to pull off next.

And when you're done, call `va_end()` to wrap it up. You **must** (the spec says) call this on a particular `va_list` variable before you decide to call either `va_start()` or `va_copy()` on it again. I know we haven't talked about `va_copy()` yet.

So the standard progression is:

- `va_start()` to initialize your `va_list` variable
- Repeatedly `va_arg()` to get the values
- `va_end()` to deinitialize your `va_list` variable

I also mentioned `va_copy()` up there; it makes a copy of your `va_list` variable in the exact same state. That is, if you haven't started with `va_arg()` with the source variable, the new one won't be started, either. If you've consumed 5 variables with `va_arg()` so far, the copy will also reflect that.

[1] That is, us lowly developers aren't supposed to know what's in there or what it means. The spec doesn't dictate what it is in detail.

[2] Honestly, it would be possible to remove that limitation from the language, but the idea is that the macros `va_start()`, `va_arg()`, and `va_end()` should be able to be written in C. And to make that happen, we need some way to initialize a pointer to the location of the first parameter. And to do that, we need the *name* of the first parameter. It would require a language extension to make this possible, and so far the committee hasn't found a rationale for doing so.

`va_copy()` can be useful if you need to scan ahead through the arguments but need to also remember your current place.

25.4 Library Functions That Use `va_lists`

One of the other uses for these is pretty cool: writing your own custom `printf()` variant. It would be a pain to have to handle all those format specifiers right? All zillion of them?

Luckily, there are `printf()` variants that accept a working `va_list` as an argument. You can use these to wrap up and make your own custom `printf()`s!

These functions start with the letter `v`, such as `vprintf()`, `vfprintf()`, `vsprintf()`, and `vsnprintf()`. Basically all your `printf()` golden oldies except with a `v` in front.

Let's make a function `my_printf()` that works just like `printf()` except it takes an extra argument up front.

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  int my_printf(int serial, const char *format, ...)
5  {
6      va_list va;
7
8      // Do my custom work
9      printf("The serial number is: %d\n", serial);
10
11     // Then pass the rest off to vprintf()
12     va_start(va, format);
13     int rv = vprintf(format, va);
14     va_end(va);
15
16     return rv;
17 }
18
19 int main(void)
20 {
21     int x = 10;
22     float y = 3.2;
23
24     my_printf(3490, "x is %d, y is %f\n", x, y);
25 }
```

See what we did there? On lines 12-14 we started a new `va_list` variable, and then just passed it right into `vprintf()`. And it knows just want to do with it, because it has all the `printf()` smarts built-in.

We still have to call `va_end()` when we're done, though, so don't forget that!

Chapter 26

Locale and Internationalization

Localization is the process of making your app ready to work well in different locales (or countries).

As you might know, not everyone uses the same character for decimal points or for thousands separators… or for currency.

These locales have names, and you can select one to use. For example, a US locale might write a number like:

100,000.00

Whereas in Brazil, the same might be written with the commas and decimal points swapped:

100.000,00

Makes it easier to write your code so it ports to other nationalities with ease!

Well, sort of. Turns out C only has one built-in locale, and it's limited. The spec really leaves a lot of ambiguity here; it's hard to be completely portable.

But we'll do our best!

26.1 Setting the Localization, Quick and Dirty

For these calls, include `<locale.h>`.

There is basically one thing you can portably do here in terms of declaring a specific locale. This is likely what you want to do if you're going to do locale anything:

```
setlocale(LC_ALL, "");  // Use this environment's locale for everything
```

You'll want to call that so that the program gets initialized with your current locale.

Getting into more details, there is one more thing you can do and stay portable:

```
setlocale(LC_ALL, "C");  // Use the default C locale
```

but that's called by default every time your program starts, so there's not much need to do it yourself.

In that second string, you can specify any locale supported by your system. This is completely system-dependent, so it will vary. On my system, I can specify this:

```
setlocale(LC_ALL, "en_US.UTF-8");  // Non-portable!
```

And that'll work. But it's only portable to systems which have that exact same name for that exact same locale, and you can't guarantee it.

By passing in an empty string ("") for the second argument, you're telling C, ``Hey, figure out what the current locale on this system is so I don't have to tell you."

26.2 Getting the Monetary Locale Settings

Because moving green pieces of paper around promises to be the key to happiness[1], let's talk about monetary locale. When you're writing portable code, you have to know what to type for cash, right? Whether that's ``$", ``€", ``¥", or ``£".

How can you write that code without going insane? Luckily, once you call `setlocale(LC_ALL, "")`, you can just look these up with a call to `localeconv()`:

```
struct lconv *x = localeconv();
```

This function returns a pointer to a statically-allocated `struct lconv` that has all that juicy information you're looking for.

Here are the fields of `struct lconv` and their meanings.

First, some conventions. An _p_ means ``positive", and _n_ means ``negative", and `int_` means ``international". Though a lot of these are type `char` or `char*`, most (or the strings they point to) are actually treated as integers[2].

Before we go further, know that `CHAR_MAX` (from `<limits.h>`) is the maximum value that can be held in a `char`. And that many of the following `char` values use that to indicate the value isn't available in the given locale.

Field	Description
`char *mon_decimal_point`	Decimal pointer character for money, e.g. `"."`.
`char *mon_thousands_sep`	Thousands separator character for money, e.g. `","`.
`char *mon_grouping`	Grouping description for money (see below).
`char *positive_sign`	Positive sign for money, e.g. `"+"` or `""`.
`char *negative_sign`	Negative sign for money, e.g. `"-"`.
`char *currency_symbol`	Currency symbol, e.g. `"$"`.
`char frac_digits`	When printing monetary amounts, how many digits to print past the decimal point, e.g. `2`.
`char p_cs_precedes`	`1` if the `currency_symbol` comes before the value for a non-negative monetary amount, `0` if after.
`char n_cs_precedes`	`1` if the `currency_symbol` comes before the value for a negative monetary amount, `0` if after.
`char p_sep_by_space`	Determines the separation of the `currency symbol` from the value for non-negative amounts (see below).
`char n_sep_by_space`	Determines the separation of the `currency symbol` from the value for negative amounts (see below).
`char p_sign_posn`	Determines the `positive_sign` position for non-negative values.
`char p_sign_posn`	Determines the `positive_sign` position for negative values.
`char *int_curr_symbol`	International currency symbol, e.g. `"USD "`.
`char int_frac_digits`	International value for `frac_digits`.
`char int_p_cs_precedes`	International value for `p_cs_precedes`.
`char int_n_cs_precedes`	International value for `n_cs_precedes`.
`char int_p_sep_by_space`	International value for `p_sep_by_space`.

[1] ``This planet has---or rather had---a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movement of small green pieces of paper, which was odd because on the whole it wasn't the small green pieces of paper that were unhappy." ---The Hitchhiker's Guide to the Galaxy, Douglas Adams

[2] Remember that `char` is just a byte-sized integer.

Field	Description
`char int_n_sep_by_space`	International value for `n_sep_by_space`.
`char int_p_sign_posn`	International value for `p_sign_posn`.
`char int_n_sign_posn`	International value for `n_sign_posn`.

26.2.1 Monetary Digit Grouping

OK, this is a trippy one. `mon_grouping` is a `char*`, so you might be thinking it's a string. But in this case, no, it's really an array of `chars`. It should always end either with a `0` or `CHAR_MAX`.

These values describe how to group sets of numbers in currency to the *left* of the decimal (the whole number part).

For example, we might have:

```
  2   1   0
 --- --- ---
$100,000,000.00
```

These are groups of three. Group 0 (just left of the decimal) has 3 digits. Group 1 (next group to the left) has 3 digits, and the last one also has 3.

So we could describe these groups, from the right (the decimal) to the left with a bunch of integer values representing the group sizes:

```
3 3 3
```

And that would work for values up to $100,000,000.

But what if we had more? We could keep adding `3`s…

```
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

but that's crazy. Luckily, we can specify `0` to indicate that the previous group size repeats:

```
3 0
```

Which means to repeat every 3. That would handle $100, $1,000, $10,000, $10,000,000, $100,000,000,000, and so on.

You can go legitimately crazy with these to indicate some weird groupings.

For example:

```
4 3 2 1 0
```

would indicate:

```
$1,0,0,0,0,00,000,0000.00
```

One more value that can occur is `CHAR_MAX`. This indicates that no more grouping should occur, and can appear anywhere in the array, including the first value.

```
3 2 CHAR_MAX
```

would indicate:

```
100000000,00,000.00
```

for example.

And simply having `CHAR_MAX` in the first array position would tell you there was to be no grouping at all.

26.2.2 Separators and Sign Position

All the `sep_by_space` variants deal with spacing around the currency sign. Valid values are:

Value	Description
`0`	No space between currency symbol and value.
`1`	Separate the currency symbol (and sign, if any) from the value with a space.
`2`	Separate the sign symbol from the currency symbol (if adjacent) with a space, otherwise separate the sign symbol from the value with a space.

The `sign_posn` variants are determined by the following values:

Value	Description
`0`	Put parens around the value and the currency symbol.
`1`	Put the sign string in front of the currency symbol and value.
`2`	Put the sign string after the currency symbol and value.
`3`	Put the sign string directly in front of the currency symbol.
`4`	Put the sign string directly behind the currency symbol.

26.2.3 Example Values

When I get the values on my system, this is what I see (grouping string displayed as individual byte values):

```
mon_decimal_point  = "."
mon_thousands_sep  = ","
mon_grouping       = 3 3 0
positive_sign      = ""
negative_sign      = "-"
currency_symbol    = "$"
frac_digits        = 2
p_cs_precedes      = 1
n_cs_precedes      = 1
p_sep_by_space     = 0
n_sep_by_space     = 0
p_sign_posn        = 1
n_sign_posn        = 1
int_curr_symbol    = "USD "
int_frac_digits    = 2
int_p_cs_precedes  = 1
int_n_cs_precedes  = 1
int_p_sep_by_space = 1
int_n_sep_by_space = 1
int_p_sign_posn    = 1
int_n_sign_posn    = 1
```

26.3 Localization Specifics

Notice how we passed the macro `LC_ALL` to `setlocale()` earlier… this hints that there might be some variant that allows you to be more precise about which *parts* of the locale you're setting.

Let's take a look at the values you can see for these:

Macro	Description
`LC_ALL`	Set all of the following to the given locale.
`LC_COLLATE`	Controls the behavior of the `strcoll()` and `strxfrm()` functions.
`LC_CTYPE`	Controls the behavior of the character-handling functions[3].
`LC_MONETARY`	Controls the values returned by `localeconv()`.
`LC_NUMERIC`	Controls the decimal point for the `printf()` family of functions.
`LC_TIME`	Controls time formatting of the `strftime()` and `wcsftime()` time and date printing functions.

It's pretty common to see `LC_ALL` being set, but, hey, at least you have options.

Also I should point out that `LC_CTYPE` is one of the biggies because it ties into wide characters, a significant can of worms that we'll talk about later.

[3]Except for `isdigit()` and `isxdigit()`.

Chapter 27

Unicode, Wide Characters, and All That

Before we begin, note that this is an active area of language development in C as it works to get past some, erm, *growing pains*. When C2x comes out, updates here are probable.

Most people are basically interested in the deceptively simple question, ``How do I use such-and-such character set in C?'' We'll get to that. But as we'll see, it might already work on your system. Or you might have to punt to a third-party library.

We're going to talk about a lot of things this chapter---some are platform agnostic, and some are C-specific.

Let's get an outline first of what we're going to look at:

- Unicode background
- Character encoding background
- Source and Execution character Sets
- Using Unicode and UTF-8
- Using other character types like `wchar_t`, `char16_t`, and `char32_t`

Let's dive in!

27.1 What is Unicode?

Back in the day, it was popular in the US and much of the world to use a 7-bit or 8-bit encoding for characters in memory. This meant we could have 128 or 256 characters (including non-printable characters) total. That was fine for a US-centric world, but it turns out there are actually other alphabets out there---who knew? Chinese has over 50,000 characters, and that's not fitting in a byte.

So people came up with all kinds of alternate ways to represent their own custom character sets. And that was fine, but turned into a compatibility nightmare.

To escape it, Unicode was invented. One character set to rule them all. It extends off into infinity (effectively) so we'll never run out of space for new characters. It has Chinese, Latin, Greek, cuneiform, chess symbols, emojis… just about everything, really! And more is being added all the time!

27.2 Code Points

I want to talk about two concepts here. It's confusing because they're both numbers… different numbers for the same thing. But bear with me.

Let's loosely define *code point* to mean a numeric value representing a character. (Code points can also represent unprintable control characters, but just assume I mean something like the letter ``B'' or the character

``π".)

Each code point represents a unique character. And each character has a unique numeric code point associated with it.

For example, in Unicode, the numeric value 66 represents ``B", and 960 represents ``π". Other character mappings that aren't Unicode use different values, potentially, but let's forget them and concentrate on Unicode, the future!

So that's one thing: there's a number that represents each character. In Unicode, these numbers run from 0 to over 1 million.

Got it?

Because we're about to flip the table a little.

27.3 Encoding

If you recall, an 8-bit byte can hold values from 0-255, inclusive. That's great for ``B" which is 66---that fits in a byte. But ``π" is 960, and that doesn't fit in a byte! We need another byte. How do we store all that in memory? Or what about bigger numbers, like 195,024? That's going to need a number of bytes to hold.

The Big Question: how are these numbers represented in memory? This is what we call the *encoding* of the characters.

So we have two things: one is the code point which tells us effectively the serial number of a particular character. And we have the encoding which tells us how we're going to represent that number in memory.

There are plenty of encodings. You can make up your own right now, if you want[1]. But we're going to look at some really common encodings that are in use with Unicode.

Encoding	Description
UTF-8	A byte-oriented encoding that uses a variable number of bytes per character. This is the one to use.
UTF-16	A 16-bit per character[2] encoding.
UTF-32	A 32-bit per character encoding.

With UTF-16 and UTF-32, the byte order matters, so you might see UTF-16BE for big-endian and UTF-16LE for little-endian. Same for UTF-32. Technically, if unspecified, you should assume big-endian. But since Windows uses UTF-16 extensively and is little-endian, sometimes that is assumed[3].

Let's look at some examples. I'm going to write the values in hex because that's exactly two digits per 8-bit byte, and it makes it easier to see how things are arranged in memory.

Character	Code Point	UTF-16BE	UTF-32BE	UTF-16LE	UTF-32LE	UTF-8
A	41	0041	00000041	4100	41000000	41
B	42	0042	00000042	4200	42000000	42
~	7E	007E	0000007E	7E00	7E000000	7E
π	3C0	03C0	000003C0	C003	C0030000	CF80
€	20AC	20AC	000020AC	AC20	AC200000	E282AC

[1] For example, we could store the code point in a big-endian 32-bit integer. Straightforward! We just invented an encoding! Actually not; that's what UTF-32BE encoding is. Oh well---back to the grind!

[2] Ish. Technically, it's variable width---there's a way to represent code points higher than 2^{16} by putting two UTF-16 characters together.

[3] There's a special character called the *Byte Order Mark* (BOM), code point 0xFEFF, that can optionally precede the data stream and indicate the endianess. It is not required, however.

Look in there for the patterns. Note that UTF-16BE and UTF-32BE are simply the code point represented directly as 16- and 32-bit values[4].

Little-endian is the same, except the bytes are in little-endian order.

Then we have UTF-8 at the end. First you might notice that the single-byte code points are represented as a single byte. That's nice. You might also notice that different code points take different number of bytes. This is a variable-width encoding.

So as soon as we get above a certain value, UTF-8 starts using additional bytes to store the values. And they don't appear to correlate with the code point value, either.

The details of UTF-8 encoding[5] are beyond the scope of this guide, but it's enough to know that it has a variable number of bytes per code point, and those byte values don't match up with the code point *except for the first 128 code points*. If you really want to learn more, Computerphile has a great UTF-8 video with Tom Scott[6].

That last bit is a neat thing about Unicode and UTF-8 from a North American perspective: it's backward compatible with 7-bit ASCII encoding! So if you're used to ASCII, UTF-8 is the same! Every ASCII-encoded document is also UTF-8 encoded! (But not the other way around, obviously.)

It's probably that last point more than any other that is driving UTF-8 to take over the world.

27.4 Source and Execution Character Sets

When programming in C, there are (at least) three character sets that are in play:

- The one that your code exists on disk as.
- The one the compiler translates that into just as compilation begins (the *source character set*). This might be the same as the one on disk, or it might not.
- The one the compiler translates the source character set into for execution (the *execution character set*). This might be the same as the source character set, or it might not.

Your compiler probably has options to select these character sets at build-time.

The basic character set for both source and execution will contain the following characters:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
space tab vertical-tab
form-feed end-of-line
```

Those are the characters you can use in your source and remain 100% portable.

The execution character set will additionally have characters for alert (bell/flash), backspace, carriage return, and newline.

But most people don't go to that extreme and freely use their extended character sets in source and executable, especially now that Unicode and UTF-8 are getting more common. I mean, the basic character set doesn't even allow for `@`, `$`, or `` ` ``!

[4] Again, this is only true in UTF-16 for characters that fit in two bytes.

[5] https://en.wikipedia.org/wiki/UTF-8

[6] https://www.youtube.com/watch?v=MijmeoH9LT4

Notably, it's a pain (though possible with escape sequences) to enter Unicode characters using only the basic character set.

27.5 Unicode in C

Before I get into encoding in C, let's talk about Unicode from a code point standpoint. There is a way in C to specify Unicode characters and these will get translated by the compiler into the execution character set[7].

So how do we do it?

How about the euro symbol, code point 0x20AC. (I've written it in hex because both ways of representing it in C require hex.) How can we put that in our C code?

Use the `\u` escape to put it in a string, e.g. `"\u20AC"` (case for the hex doesn't matter). You must put **exactly four** hex digits after the `\u`, padding with leading zeros if necessary.

Here's an example:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

So `\u` works for 16-bit Unicode code points, but what about ones bigger than 16 bits? For that, we need capitals: `\U`.

For example:

```
char *s = "\U0001D4D1";

printf("%s\n", s);  // Prints a mathematical letter "B"
```

It's the same as `\u`, just with 32 bits instead of 16. These are equivalent:

```
\u03C0
\U000003C0
```

Again, these are translated into the execution character set during compilation. They represent Unicode code points, not any specific encoding. Furthermore, if a Unicode code point is not representable in the execution character set, the compiler can do whatever it wants with it.

Now, you might wonder why you can't just do this:

```
char *s = "€1.23";

printf("%s\n", s);  // €1.23
```

And you probably can, given a modern compiler. The source character set will be translated for you into the execution character set by the compiler. But compilers are free to puke out if they find any characters that aren't included in their extended character set, and the € symbol certainly isn't in the basic character set.

Caveat from the spec: you can't use `\u` or `\U` to encode any code points below 0xA0 except for 0x24 (`$`), 0x40 (`@`), and 0x60 (`` ` ``)---yes, those are precisely the trio of common punctuation marks missing from the basic character set. Apparently this restriction is relaxed in the upcoming version of the spec.

Finally, you can also use these in identifiers in your code, with some restrictions. But I don't want to get into that here. We're all about string handling in this chapter.

And that's about it for Unicode in C (except encoding).

[7] Presumably the compiler makes the best effort to translate the code point to whatever the output encoding is, but I can't find any guarantees in the spec.

27.6 A Quick Note on UTF-8 Before We Swerve into the Weeds

It could be that your source file on disk, the extended source characters, and the extended execution characters are all in UTF-8 format. And the libraries you use expect UTF-8. This is the glorious future of UTF-8 everywhere.

If that's the case, and you don't mind being non-portable to systems that aren't like that, then just run with it. Stick Unicode characters in your source and data at will. Use regular C strings and be happy.

A lot of things will just work (albeit non-portably) because UTF-8 strings can safely be NUL-terminated just like any other C string. But maybe losing portability in exchange for easier character handling is a tradeoff that's worth it to you.

There are some caveats, however:

- Things like `strlen()` report the number of bytes in a string, not the number of characters, necessarily. (The `mbstowcs()` returns the number of characters in a string when you convert it to wide characters. POSIX extends this so you can pass `NULL` for the first argument if you just want the character count.)
- The following won't work properly with characters of more than one byte: `strtok()`, `strchr()` (use `strstr()` instead), `strspn()`-type functions, `toupper()`, `tolower()`, `isalpha()`-type functions, and probably more. Beware anything that operates on bytes.
- `printf()` variants allow for a way to only print so many bytes of a string[8]. You want to make certain you print the correct number of bytes to end on a character boundary.
- If you want to `malloc()` space for a string, or declare an array of `chars` for one, be aware that the maximum size could be more than you were expecting. Each character could take up to `MB_LEN_MAX` bytes (from `<limits.h>`)---except characters in the basic character set which are guaranteed to be one byte.

And probably others I haven't discovered. Let me know what pitfalls there are out there…

27.7 Different Character Types

I want to introduce more character types. We're used to `char`, right?

But that's too easy. Let's make things a lot more difficult! Yay!

27.7.1 Multibyte Characters

First of all, I want to potentially change your thinking about what a string (array of `chars`) is. These are *multibyte strings* made up of *multibyte characters*.

That's right---your run-of-the-mill string of characters is multibyte. When someone says ``C string", they mean ``C multibyte string".

Even if a particular character in the string is only a single byte, or if a string is made up of only single characters, it's known as a multibyte string.

For example:

```
char c[128] = "Hello, world!";  // Multibyte string
```

What we're saying here is that a particular character that's not in the basic character set could be composed of multiple bytes. Up to `MB_LEN_MAX` of them (from `<limits.h>`). Sure, it only looks like one character on the screen, but it could be multiple bytes.

You can throw Unicode values in there, as well, as we saw earlier:

[8] With a format specifier like `"%.12s"`, for example.

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

But here we're getting into some weirdness, because check this out:

```
char *s = "\u20AC1.23";  // €1.23

printf("%zu\n", strlen(s));  // 7!
```

The string length of `"€1.23"` is `7`?! Yes! Well, on my system, yes! Remember that `strlen()` returns the number of bytes in the string, not the number of characters. (When we get to ``wide characters", coming up, we'll see a way to get the number of characters in the string.)

Note that while C allows individual multibyte `char` constants (as opposed to `char*`), the behavior of these varies by implementation and your compiler might warn on it.

GCC, for example, warns of multi-character character constants for the following two lines (and, on my system, prints out the UTF-8 encoding):

```
printf("%x\n", '€');
printf("%x\n", '\u20ac');
```

27.7.2 Wide Characters

If you're not a multibyte character, then you're a *wide character*.

A wide character is a single value that can uniquely represent any character in the current locale. It's analogous to Unicode code points. But it might not be. Or it might be.

Basically, where multibyte character strings are arrays of bytes, wide character strings are arrays of *characters*. So you can start thinking on a character-by-character basis rather than a byte-by-byte basis (the latter of which gets all messy when characters start taking up variable numbers of bytes).

Wide characters can be represented by a number of types, but the big standout one is `wchar_t`. It's the main one. It's like `char`, except wide.

You might be wondering if you can't tell if it's Unicode or not, how does that allow you much flexibility in terms of writing code? `wchar_t` opens some of those doors, as there are a rich set of functions you can use to deal with `wchar_t` strings (like getting the length, etc.) without caring about the encoding.

27.8 Using Wide Characters and `wchar_t`

Time for a new type: `wchar_t`. This is the main wide character type. Remember how a `char` is only one byte? And a byte's not enough to represent all characters, potentially? Well, this one is enough.

To use `wchar_t`, include `<wchar.h>`.

How many bytes big is it? Well, it's not totally clear. Could be 16 bits. Could be 32 bits.

But wait, you're saying---if it's only 16 bits, it's not big enough to hold all the Unicode code points, is it? You're right---it's not. The spec doesn't require it to be. It just has to be able to represent all the characters in the current locale.

This can cause grief with Unicode on platforms with 16-bit `wchar_t`s (ahem---Windows). But that's out of scope for this guide.

You can declare a string or character of this type with the `L` prefix, and you can print them with the `%ls` (``ell ess") format specifier. Or print an individual `wchar_t` with `%lc`.

```
wchar_t *s = L"Hello, world!";
wchar_t c = L'B';

printf("%ls %lc\n", s, c);
```

Now---are those characters stored as Unicode code points, or not? Depends on the implementation. But you can test if they are with the macro `__STDC_ISO_10646__`. If this is defined, the answer is, ``It's Unicode!''

More detailedly, the value in that macro is an integer in the form `yyyymm` that lets you know what Unicode standard you can rely on---whatever was in effect on that date.

But how do you use them?

27.8.1 Multibyte to `wchar_t` Conversions

So how do we get from the byte-oriented standard strings to the character-oriented wide strings and back?

We can use a couple string conversion functions to make this happen.

First, some naming conventions you'll see in these functions:

- `mb`: multibyte
- `wc`: wide character
- `mbs`: multibyte string
- `wcs`: wide character string

So if we want to convert a multibyte string to a wide character string, we can call the `mbstowcs()`. And the other way around: `wcstombs()`.

Conversion Function	Description
`mbtowc()`	Convert a multibyte character to a wide character.
`wctomb()`	Convert a wide character to a multibyte character.
`mbstowcs()`	Convert a multibyte string to a wide string.
`wcstombs()`	Convert a wide string to a multibyte string.

Let's do a quick demo where we convert a multibyte string to a wide character string, and compare the string lengths of the two using their respective functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Get out of the C locale to one that likely has the euro symbol
    setlocale(LC_ALL, "");

    // Original multibyte string with a euro symbol (Unicode point 20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Wide character array that will hold the converted string
    wchar_t wc_string[128];  // Holds up to 128 wide characters

```

```
    // Convert the MB string to WC; this returns the number of wide chars
    size_t wc_len = mbstowcs(wc_string, mb_string, 128);

    // Print result--note the %ls for wide char strings
    printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
    printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
}
```

On my system, this outputs:

```
multibyte: "The cost is €1.23" (19 bytes)
wide char: "The cost is €1.23" (17 characters)
```

(Your system might vary on the number of bytes depending on your locale.)

One interesting thing to note is that `mbstowcs()`, in addition to converting the multibyte string to wide, returns the length (in characters) of the wide character string. On POSIX-compliant systems, you can take advantage of a special mode where it *only* returns the length-in-characters of a given multibyte string: you just pass `NULL` to the destination, and `0` to the maximum number of characters to convert (this value is ignored).

(In the code below, I'm using my extended source character set---you might have to replace those with `\u` escapes.)

```
setlocale(LC_ALL, "");

// The following string has 7 characters
size_t len_in_chars = mbstowcs(NULL, "§¶°±π€•", 0);

printf("%zu", len_in_chars);  // 7
```

Again, that's a non-portable POSIX extension.

And, of course, if you want to convert the other way, it's `wcstombs()`.

27.9 Wide Character Functionality

Once we're in wide character land, we have all kinds of functionality at our disposal. I'm just going to summarize a bunch of the functions here, but basically what we have here are the wide character versions of the multibyte string functions that we're use to. (For example, we know `strlen()` for multibyte strings; there's a `wcslen()` for wide character strings.)

27.9.1 `wint_t`

A lot of these functions use a `wint_t` to hold single characters, whether they are passed in or returned.

It is related to `wchar_t` in nature. A `wint_t` is an integer that can represent all values in the extended character set, and also a special end-of-file character, `WEOF`.

This is used by a number of single-character-oriented wide character functions.

27.9.2 I/O Stream Orientation

The tl;dr here is to not mix and match byte-oriented functions (like `fprintf()`) with wide-oriented functions (like `fwprintf()`). Decide if a stream will be byte-oriented or wide-oriented and stick with those types of I/O functions.

In more detail: streams can be either byte-oriented or wide-oriented. When a stream is first created, it has no orientation, but the first read or write will set the orientation.

If you first use a wide operation (like `fwprintf()`) it will orient the stream wide.

If you first use a byte operation (like `fprintf()`) it will orient the stream by bytes.

You can manually set an unoriented stream one way or the other with a call to `fwide()`. You can use that same function to get the orientation of a stream.

If you need to change the orientation mid-flight, you can do it with `freopen()`.

27.9.3 I/O Functions

Typically include `<stdio.h>` and `<wchar.h>` for these.

I/O Function	Description
`wprintf()`	Formatted console output.
`wscanf()`	Formatted console input.
`getwchar()`	Character-based console input.
`putwchar()`	Character-based console output.
`fwprintf()`	Formatted file output.
`fwscanf()`	Formatted file input.
`fgetwc()`	Character-based file input.
`fputwc()`	Character-based file output.
`fgetws()`	String-based file input.
`fputws()`	String-based file output.
`swprintf()`	Formatted string output.
`swscanf()`	Formatted string input.
`vfwprintf()`	Variadic formatted file output.
`vfwscanf()`	Variadic formatted file input.
`vswprintf()`	Variadic formatted string output.
`vswscanf()`	Variadic formatted string input.
`vwprintf()`	Variadic formatted console output.
`vwscanf()`	Variadic formatted console input.
`ungetwc()`	Push a wide character back on an output stream.
`fwide()`	Get or set stream multibyte/wide orientation.

27.9.4 Type Conversion Functions

Typically include `<wchar.h>` for these.

Conversion Function	Description
`wcstod()`	Convert string to `double`.
`wcstof()`	Convert string to `float`.
`wcstold()`	Convert string to `long double`.
`wcstol()`	Convert string to `long`.
`wcstoll()`	Convert string to `long long`.
`wcstoul()`	Convert string to `unsigned long`.
`wcstoull()`	Convert string to `unsigned long long`.

27.9.5 String and Memory Copying Functions

Typically include `<wchar.h>` for these.

Copying Function	Description
`wcscpy()`	Copy string.
`wcsncpy()`	Copy string, length-limited.
`wmemcpy()`	Copy memory.
`wmemmove()`	Copy potentially-overlapping memory.
`wcscat()`	Concatenate strings.
`wcsncat()`	Concatenate strings, length-limited.

27.9.6 String and Memory Comparing Functions

Typically include `<wchar.h>` for these.

Comparing Function	Description
`wcscmp()`	Compare strings lexicographically.
`wcsncmp()`	Compare strings lexicographically, length-limited.
`wcscoll()`	Compare strings in dictionary order by locale.
`wmemcmp()`	Compare memory lexicographically.
`wcsxfrm()`	Transform strings into versions such that `wcscmp()` behaves like `wcscoll()`[9].

27.9.7 String Searching Functions

Typically include `<wchar.h>` for these.

Searching Function	Description
`wcschr()`	Find a character in a string.
`wcsrchr()`	Find a character in a string from the back.
`wmemchr()`	Find a character in memory.
`wcsstr()`	Find a substring in a string.
`wcspbrk()`	Find any of a set of characters in a string.
`wcsspn()`	Find length of substring including any of a set of characters.
`wcscspn()`	Find length of substring before any of a set of characters.
`wcstok()`	Find tokens in a string.

27.9.8 Length/Miscellaneous Functions

Typically include `<wchar.h>` for these.

Length/Misc Function	Description
`wcslen()`	Return the length of the string.
`wmemset()`	Set characters in memory.
`wcsftime()`	Formatted date and time output.

[9] `wcscoll()` is the same as `wcsxfrm()` followed by `wcscmp()`.

27.9.9 Character Classification Functions

Include `<wctype.h>` for these.

Length/Misc Function	Description
`iswalnum()`	True if the character is alphanumeric.
`iswalpha()`	True if the character is alphabetic.
`iswblank()`	True if the character is blank (space-ish, but not a newline).
`iswcntrl()`	True if the character is a control character.
`iswdigit()`	True if the character is a digit.
`iswgraph()`	True if the character is printable (except space).
`iswlower()`	True if the character is lowercase.
`iswprint()`	True if the character is printable (including space).
`iswpunct()`	True if the character is punctuation.
`iswspace()`	True if the character is whitespace.
`iswupper()`	True if the character is uppercase.
`iswxdigit()`	True if the character is a hex digit.
`towlower()`	Convert character to lowercase.
`towupper()`	Convert character to uppercase.

27.10 Parse State, Restartable Functions

We're going to get a little bit into the guts of multibyte conversion, but this is a good thing to understand, conceptually.

Imagine how your program takes a sequence of multibyte characters and turns them into wide characters, or vice-versa. It might, at some point, be partway through parsing a character, or it might have to wait for more bytes before it makes the determination of the final value.

This parse state is stored in an opaque variable of type `mbstate_t` and is used every time conversion is performed. That's how the conversion functions keep track of where they are mid-work.

And if you change to a different character sequence mid-stream, or try to seek to a different place in your input sequence, it could get confused over that.

Now you might want to call me on this one: we just did some conversions, above, and I never mentioned any `mbstate_t` anywhere.

That's because the conversion functions like `mbstowcs()`, `wctomb()`, etc. each have their own `mbstate_t` variable that they use. There's only one per function, though, so if you're writing multithreaded code, they're not safe to use.

Fortunately, C defines *restartable* versions of these functions where you can pass in your own `mbstate_t` on per-thread basis if you need to. If you're doing multithreaded stuff, use these!

Quick note on initializing an `mbstate_t` variable: just `memset()` it to zero. There is no built-in function to force it to be initialized.

```
mbstate_t mbs;

// Set the state to the initial state
memset(&mbs, 0, sizeof mbs);
```

Here is a list of the restartable conversion functions---note the naming convension of putting an ``r'' after the ``from'' type:

- `mbrtowc()`---multibyte to wide character
- `wcrtomb()`---wide character to multibyte
- `mbsrtowcs()`---multibyte string to wide character string
- `wcsrtombs()`---wide character string to multibyte string

These are really similar to their non-restartable counterparts, except they require you pass in a pointer to your own `mbstate_t` variable. And also they modify the source string pointer (to help you out if invalid bytes are found), so it might be useful to save a copy of the original.

Here's the example from earlier in the chapter reworked to pass in our own `mbstate_t`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Get out of the C locale to one that likely has the euro symbol
    setlocale(LC_ALL, "");

    // Original multibyte string with a euro symbol (Unicode point 20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Wide character array that will hold the converted string
    wchar_t wc_string[128];  // Holds up to 128 wide characters

    // Set up the conversion state
    mbstate_t mbs;
    memset(&mbs, 0, sizeof mbs);  // Initial state

    // mbsrtowcs() modifies the input pointer to point at the first
    // invalid character, or NULL if successful. Let's make a copy of
    // the pointer for mbsrtowcs() to mess with so our original is
    // unchanged.
    //
    // This example will probably be successful, but we check farther
    // down to see.
    const char *invalid = mb_string;

    // Convert the MB string to WC; this returns the number of wide chars
    size_t wc_len = mbsrtowcs(wc_string, &invalid, 128, &mbs);

    if (invalid == NULL) {
        printf("No invalid characters found\n");

        // Print result--note the %ls for wide char strings
        printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
        printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
    } else {
        ptrdiff_t offset = invalid - mb_string;
        printf("Invalid character at offset %td\n", offset);
```

```
    }
}
```

For the conversion functions that manage their own state, you can reset their internal state to the initial one by passing in `NULL` for their `char*` arguments, for example:

```
mbstowcs(NULL, NULL, 0);   // Reset the parse state for mbstowcs()
mbstowcs(dest, src, 100);  // Parse some stuff
```

For I/O, each wide stream manages its own `mbstate_t` and uses that for input and output conversions as it goes.

And some of the byte-oriented I/O functions like `printf()` and `scanf()` keep their own internal state while doing their work.

Finally, these restartable conversion functions do actually have their own internal state if you pass in `NULL` for the `mbstate_t` parameter. This makes them behave more like their non-restartable counterparts.

27.11 Unicode Encodings and C

In this section, we'll see what C can (and can't) do when it comes to three specific Unicode encodings: UTF-8, UTF-16, and UTF-32.

27.11.1 UTF-8

To refresh before this section, read the UTF-8 quick note, above.

Aside from that, what are C's UTF-8 capabilities?

Well, not much, unfortunately.

You can tell C that you specifically want a string literal to be UTF-8 encoded, and it'll do it for you. You can prefix a string with `u8`:

```
char *s = u8"Hello, world!";

printf("%s\n", s);   // Hello, world!--if you can output UTF-8
```

Now, can you put Unicode characters in there?

```
char *s = u8"€123";
```

Sure! If the extended source character set supports it. (gcc does.)

What if it doesn't? You can specify a Unicode code point with your friendly neighborhood `\u` and `\U`, as noted above.

But that's about it. There's no portable way in the standard library to take arbirary input and turn it into UTF-8 unless your locale is UTF-8. Or to parse UTF-8 unless your locale is UTF-8.

So if you want to do it, either be in a UTF-8 locale and:

```
setlocale(LC_ALL, "");
```

or figure out a UTF-8 locale name on your local machine and set it explicitly like so:

```
setlocale(LC_ALL, "en_US.UTF-8");  // Non-portable name
```

Or use a third-party library.

27.11.2 UTF-16, UTF-32, `char16_t`, and `char32_t`

`char16_t` and `char32_t` are a couple other potentially wide character types with sizes of 16 bits and 32 bits, respectively. Not necessarily wide, because if they can't represent every character in the current locale, they lose their wide character nature. But the spec refers them as ``wide character'' types all over the place, so there we are.

These are here to make things a little more Unicode-friendly, potentially.

To use, include `<uchar.h>`. (That's ``u'', not ``w''.)

This header file doesn't exist on OS X---bummer. If you just want the types, you can:

```
#include <stdint.h>

typedef int_least16_t char16_t;
typedef int_least32_t char32_t;
```

But if you also want the functions, that's all on you.

Assuming you're still good to go, you can declare a string or character of these types with the `u` and `U` prefixes:

```
char16_t *s = u"Hello, world!";
char16_t c = u'B';

char32_t *t = U"Hello, world!";
char32_t d = U'B';
```

Now---are values in these stored in UTF-16 or UTF-32? Depends on the implementation.

But you can test to see if they are. If the macros `__STDC_UTF_16__` or `__STDC_UTF_32__` are defined (to `1`) it means the types hold UTF-16 or UTF-32, respectively.

If you're curious, and I know you are, the values, if UTF-16 or UTF-32, are stored in the native endianess. That is, you should be able to compare them straight up to Unicode code point values:

```
char16_t pi = u"\u03C0";  // pi symbol

#if __STDC_UTF_16__
pi == 0x3C0;  // Always true
#else
pi == 0x3C0;  // Probably not true
#endif
```

27.11.3 Multibyte Conversions

You can convert from your multibyte encoding to `char16_t` or `char32_t` with a number of helper functions.

(Like I said, though, the result might not be UTF-16 or UTF-32 unless the corresponding macro is set to `1`.)

All of these functions are restartable (i.e. you pass in your own `mbstate_t`), and all of them operate character by character[10].

Conversion Function	Description
`mbrtoc16()`	Convert a multibyte character to a `char16_t` character.
`mbrtoc32()`	Convert a multibyte character to a `char32_t` character.

[10] Ish---things get funky with multi-`char16_t` UTF-16 encodings.

Conversion Function	Description
`c16rtomb()`	Convert a `char16_t` character to a multibyte character.
`c32rtomb()`	Convert a `char32_t` character to a multibyte character.

27.11.4 Third-Party Libraries

For heavy-duty conversion between different specific encodings, there are a couple mature libraries worth checking out. Note that I haven't used either of these.

- iconv[11]---Internationalization Conversion, a common POSIX-standard API available on the major platforms.
- ICU[12]---International Components for Unicode. At least one blogger found this easy to use.

If you have more noteworthy libraries, let me know.

[11] https://en.wikipedia.org/wiki/Iconv
[12] http://site.icu-project.org/

Chapter 28

Exiting a Program

Turns out there are a lot of ways to do this, and even ways to set up ``hooks" so that a function runs when a program exits.

In this chapter we'll dive in and check them out.

We already covered the meaning of the exit status code in the Exit Status section, so jump back there and review if you have to.

All the functions in this section are in `<stdlib.h>`.

28.1 Normal Exits

We'll start with the regular ways to exit a program, and then jump to some of the rarer, more esoteric ones.

When you exit a program normally, all open I/O streams are flushed and temporary files removed. Basically it's a nice exit where everything gets cleaned up and handled. It's what you want to do almost all the time unless you have reasons to do otherwise.

28.1.1 Returning From `main()`

If you've noticed, `main()` has a return type of `int`… and yet I've rarely, if ever, been `return`ing anything from `main()` at all.

This is because for `main()` only (and I can't stress enough this special case *only* applies to `main()` and no other functions anywhere) has an *implicit* `return 0` if you fall off the end.

You can explicitly `return` from `main()` any time you want, and some programmers feel it's more *Right* to always have a `return` at the end of `main()`. But if you leave it off, C will put one there for you.

So… here are the `return` rules for `main()`:

- You can return an exit status from `main()` with a `return` statement. `main()` is the only function with this special behavior. Using `return` in any other function just returns from that function to the caller.
- If you don't explicitly `return` and just fall off the end of `main()`, it's just as if you'd returned `0` or `EXIT_SUCCESS`.

28.1.2 `exit()`

This one has also made an appearance a few times. If you call `exit()` from anywhere in your program, it will exit at that point.

The argument you pass to `exit()` is the exit status.

28.1.3 Setting Up Exit Handlers with `atexit()`

You can register functions to be called when a program exits whether by returning from `main()` or calling the `exit()` function.

A call to `atexit()` with the handler function name will get it done. You can register multiple exit handlers, and they'll be called in the reverse order of registration.

Here's an example:

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

void on_exit_1(void)
{
    printf("Exit handler 1 called!\n");
}

void on_exit_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    atexit(on_exit_1);
    atexit(on_exit_2);

    printf("About to exit...\n");
}
```

And the output is:

```
About to exit...
Exit handler 2 called!
Exit handler 1 called!
```

28.2 Quicker Exits with `quick_exit()`

This is similar to a normal exit, except:

- Open files might not be flushed.
- Temporary files might not be removed.
- `atexit()` handlers won't be called.

But there is a way to register exit handlers: call `at_quick_exit()` analogously to how you'd call `atexit()`.

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

void on_quick_exit_1(void)
{
    printf("Quick exit handler 1 called!\n");
}

void on_quick_exit_2(void)
{
```

```
    printf("Quick exit handler 2 called!\n");
}

void on_exit(void)
{
    printf("Normal exit--I won't be called!\n");
}

int main(void)
{
    at_quick_exit(on_quick_exit_1);
    at_quick_exit(on_quick_exit_2);

    atexit(on_exit);  // This won't be called

    printf("About to quick exit...\n");

    quick_exit(0);
}
```

Which gives this output:

```
About to quick exit...
Quick exit handler 2 called!
Quick exit handler 1 called!
```

It works just like `exit()`/`atexit()`, except for the fact that file flushing and cleanup might not be done.

28.3 Nuke it from Orbit: `_Exit()`

Calling `_Exit()` exits immediately, period. No on-exit callback functions are executed. Files won't be flushed. Temp files won't be removed.

Use this if you have to exit *right fargin' now*.

28.4 Exiting Sometimes: `assert()`

The `assert()` statement is used to insist that something be true, or else the program will exit.

Devs often use an assert to catch Should-Never-Happen type errors.

```
#define PI 3.14159

assert(PI > 3);   // Sure enough, it is, so carry on
```

versus:

```
goats -= 100;

assert(goats >= 0);  // Can't have negative goats
```

In that case, if I try to run it and `goats` falls under `0`, this happens:

```
goat_counter: goat_counter.c:8: main: Assertion `goats >= 0' failed.
Aborted
```

and I'm dropped back to the command line.

This isn't very user-friendly, so it's only used for things the user will never see. And often people write their own assert macros that can more easily be turned off.

28.5 Abnormal Exit: `abort()`

You can use this if something has gone horribly wrong and you want to indicate as much to the outside environment. This also won't necessarily clean up any open files, etc.

I've rarely seen this used.

Some foreshadowing about *signals*: this actually works by raising a `SIGABRT` which will end the process.

What happens after that is up to the system, but on Unix-likes, it was common to dump core[1] as the program terminated.

[1] https://en.wikipedia.org/wiki/Core_dump

Chapter 29

Signal Handling

Before we start, I'm just going to advise you to generally ignore this entire chapter and use your OS's (very likely) superior signal handling functions. Unix-likes have the `sigaction()` family of functions, and Windows has… whatever it does[1].

With that out of the way, what are signals?

29.1 What Are Signals?

A *signal* is *raised* on a variety of external events. Your program can be configured to be interrupted to *handle* the signal, and, optionally, continue where it left off once the signal has been handled.

Think of it like a function that's automatically called when one of these external events occurs.

What are these events? On your system, there are probably a lot of them, but in the C spec there are just a few:

Signal	Description
`SIGABRT`	Abnormal termination---what happens when `abort()` is called.
`SIGFPE`	Floating point exception.
`SIGILL`	Illegal instruction.
`SIGINT`	Interrupt---usually the result of `CTRL-C` being hit.
`SIGSEGV`	``Segmentation Violation'': invalid memory access.
`SIGTERM`	Termination requested.

You can set up your program to ignore, handle, or allow the default action for each of these by using the `signal()` function.

29.2 Handling Signals with `signal()`

The `signal()` call takes two parameters: the signal in question, and an action to take when that signal is raised.

The action can be one of three things:

- A pointer to a handler function.

[1] Apparently it doesn't do Unix-style signals at all deep down, and they're simulated for console apps.

- `SIG_IGN` to ignore the signal.
- `SIG_DFL` to restore the default handler for the signal.

Let's write a program that you can't `CTRL-C` out of. (Don't fret---in the following program, you can also hit `RETURN` and it'll exit.)

```
1  #include <stdio.h>
2  #include <signal.h>
3
4  int main(void)
5  {
6      char s[1024];
7
8      signal(SIGINT, SIG_IGN);    // Ignore SIGINT, caused by ^C
9
10     printf("Try hitting ^C... (hit RETURN to exit)\n");
11
12     // Wait for a line of input so the program doesn't just exit
13     fgets(s, sizeof s, stdin);
14 }
```

Check out line 8---we tell the program to ignore `SIGINT`, the interrupt signal that's raised when `CTRL-C` is hit. No matter how much you hit it, the signal remains ignored. If you comment out line 8, you'll see you can `CTRL-C` with impunity and quit the program on the spot.

29.3 Writing Signal Handlers

I mentioned you could also write a handler function that gets called when the signal is raised.

These are pretty straightforward, are also very capability-limited when it comes to the spec.

Before we start, let's look at the function prototype for the `signal()` call:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Pretty easy to read, right?

WRONG! :)

Let's take a moment to take it apart for practice.

`signal()` takes two arguments: an integer `sig` representing the signal, and a pointer `func` to the handler (the handler returns `void` and takes an `int` as an argument), highlighted below:

```
               sig          func
             |-----|  |---------------|
void (*signal(int sig, void (*func)(int)))(int);
```

Basically, we're going to pass in the signal number we're interested in catching, and we're going to pass a pointer to a function of the form:

```
void f(int x);
```

that will do the actual catching.

Now---what about the rest of that prototype? It's basically all the return type. See, `signal()` will return whatever you passed as `func` on success… so that means it's returning a pointer to a function that returns `void` and takes an `int` as an argument.

```
returned
function    indicates we're             and
returns     returning a                 that function
void        pointer to function         takes an int
|--|        |                                 |---|
void       (*signal(int sig, void (*func)(int)))(int);
```

Also, it can return `SIG_ERR` in case of an error.

Let's do an example where we make it so you have to hit `CTRL-C` twice to exit.

I want to be clear that this program engages in undefined behavior in a couple ways. But it'll probably work for you, and it's hard to come up with portable non-trivial demos.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4
5  int count = 0;
6
7  void sigint_handler(int signum)
8  {
9      // The compiler is allowed to run:
10     //
11     //   signal(signum, SIG_DFL)
12     //
13     // when the handler is called. So we reset the handler here:
14     signal(SIGINT, sigint_handler);
15
16     (void)signum;   // Get rid of unused variable warning
17
18     count++;                       // Undefined behavior
19     printf("Count: %d\n", count);  // Undefined behavior
20
21     if (count == 2) {
22         printf("Exiting!\n");      // Undefined behavior
23         exit(0);
24     }
25 }
26
27 int main(void)
28 {
29     signal(SIGINT, sigint_handler);
30
31     printf("Try hitting ^C...\n");
32
33     for(;;);  // Wait here forever
34 }
```

One of the things you'll notice is that on line 14 we reset the signal handler. This is because C has the option of resetting the signal handler to its `SIG_DFL` behavior before running your custom handler. In other words, it could be a one-off. So we reset it first thing so that we handle it again for the next one.

We're ignoring the return value from `signal()` in this case. If we'd set it to a different handler earlier, it would return a pointer to that handler, which we could get like this:

```
// old_handler is type "pointer to function that takes a single
// int parameter and returns void":

void (*old_handler)(int);

old_handler = signal(SIGINT, sigint_handler);
```

That said, I'm not sure of a common use case for this. But if you need the old handler for some reason, you can get it that way.

Quick note on line 16---that's just to tell the compiler to not warn that we're not using this variable. It's like saying, ``I know I'm not using it; you don't have to warn me."

And lastly you'll see that I've marked undefined behavior in a couple places. More on that in the next section.

29.4 What Can We Actually Do?

Turns out we're pretty limited in what we can and can't do in our signal handlers. This is one of the reasons why I say you shouldn't even bother with this and instead use your OS's signal handling instead (e.g. `sigaction()` for Unix-like systems).

Wikipedia goes so far as to say the only really portable thing you can do is call `signal()` with `SIG_IGN` or `SIG_DFL` and that's it.

Here's what we **can't** portably do:

- Call any standard library function.
  - Like `printf()`, for example.
  - I think it's probably safe to call restartable/reentrant functions, but the spec doesn't allow that liberty.
- Get or set values from a local `static`, file scope, or thread-local variable.
  - Unless it's a lock-free atomic object or…
  - You're assigning into a variable of type `volatile sig_atomic_t`.

That last bit--`sig_atomic_t`--is your ticket to getting data out of a signal handler. (Unless you want to use lock-free atomic objects, which is outside the scope of this section[2].) It's an integer type that might or might not be signed. And it's bounded by what you can put in there.

You can look at the minimum and maximum allowable values in the macros `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX`[3].

Confusingly, the spec also says you can't refer ``to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t` […]"

My read on this is that you can't read or write anything that's not a lock-free atomic object. Also you can assign to an object that's `volatile sig_atomic_t`.

But can you read from it? I honestly don't see why not, except that the spec is very pointed about mentioning assigning into. But if you have to read it and make any kind of decision based on it, you might be opening up room for some kind of race conditions.

With that in mind, we can rewrite our ``hit `CTRL-C` twice to exit" code to be a little more portable, albeit less verbose on the output.

Let's change our `SIGINT` handler to do nothing except increment a value that's of type `volatile sig_atomic_t`. So it'll count the number of `CTRL-C`s that have been hit.

[2] Confusingly, `sig_atomic_t` predates the lock-free atomics and is not the same thing.

[3] If `sig_action_t` is signed, the range will be at least `-127` to `127`. If unsigned, at least `0` to `255`.

Then in our main loop, we'll check to see if that counter is over `2`, then bail out if it is.

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

volatile sig_atomic_t count = 0;

void sigint_handler(int signum)
{
    (void)signum;                    // Unused variable warning

    signal(SIGINT, sigint_handler);  // Reset signal handler

    count++;                         // Undefined behavior
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(count < 2);
}
```

Undefined behavior again? It's my read that this is, because we have to read the value in order to increment and store it.

If we only want to postpone the exit by one hitting of `CTRL-C`, we can do that without too much trouble. But any more postponement would require some ridiculous function chaining.

What we'll do is handle it once, and the handler will reset the signal to its default behavior (that is, to exit):

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

void sigint_handler(int signum)
{
    (void)signum;                    // Unused variable warning
    signal(SIGINT, SIG_DFL);         // Reset signal handler
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(1);
}
```

Later when we look at lock-free atomic variables, we'll see a way to fix the `count` version (assuming lock-free atomic variables are available on your particular system).

This is why at the beginning, I was suggesting checking out your OS's built-in signal system as a probably-superior alternative.

29.5 Friends Don't Let Friends `signal()`

Again, use your OS's built-in signal handling or the equivalent. It's not in the spec, not as portable, but probably is far more capable. Plus your OS probably has a number of signals defined that aren't in the C spec. And it's difficult to write portable code using `signal()` anyway.

Chapter 30

Variable-Length Arrays (VLAs)

C provides a way for you to declare an array whose size is determined at runtime. This gives you the benefits of dynamic runtime sizing like you get with `malloc()`, but without needing to worry about `free()`ing the memory after.

Now, a lot of people don't like VLAs. They've been banned from the Linux kernel, for example. We'll dig into more of that rationale later.

This is an optional feature of the language. The macro `__STDC_NO_VLA__` is set to `1` if VLAs are *not* present. (They were mandatory in C99, and then became optional in C11.)

```
#if __STDC_NO_VLA__ == 1
   #error Sorry, need VLAs for this program!
#endif
```

But since neither GCC nor Clang bother to define this macro, you may get limited mileage from this.

Let's dive in first with an example, and then we'll look for the devil in the details.

30.1 The Basics

A normal array is declared with a constant size, like this:

```
int v[10];
```

But with VLAs, we can use a size determined at runtime to set the array, like this:

```
int n = 10;
int v[n];
```

Now, that looks like the same thing, and in many ways is, but this gives you the flexibility to compute the size you need, and then get an array of exactly that size.

Let's ask the user to input the size of the array, and then store the index-times-10 in each of those array elements:

```
#include <stdio.h>

int main(void)
{
    int n;

    printf("Enter a number: "); fflush(stdout);
```

```
8       scanf(" %d", &n);
9
10      int v[n];
11
12      for (int i = 0; i < n; i++)
13          v[i] = i * 10;
14
15      for (int i = 0; i < n; i++)
16          printf("v[%d] = %d\n", i, v[i]);
17  }
```

(On line 7, I have an `fflush()` that should force the line to output even though I don't have a newline at the end.)

Line 10 is where we declare the VLA---once execution gets past that line, the size of the array is set to whatever `n` was at that moment. The array length can't be changed later.

You can put an expression in the brackets, as well:

```
int v[x * 100];
```

Some restrictions:

- You can't declare a VLA at file scope, and you can't make a `static` one in block scope[1].
- You can't use an initializer list to initialize the array.

Also, entering a negative value for the size of the array invokes undefined behavior---in this universe, anyway.

30.2 `sizeof` and VLAs

We're used to `sizeof` giving us the size in bytes of any particular object, including arrays. And VLAs are no exception.

The main difference is that `sizeof` on a VLA is executed at *runtime*, whereas on a non-variably-sized variable it is computed at *compile time*.

But the usage is the same.

You can even compute the number of elements in a VLA with the usual array trick:

```
size_t num_elems = sizeof v / sizeof v[0];
```

There's a subtle and correct implication from the above line: pointer arithmetic works just like you'd expect for a regular array. So go ahead and use it to your heart's content:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int n = 5;
6       int v[n];
7
8       int *p = v;
9
10      *(p+2) = 12;
11      printf("%d\n", v[2]);  // 12
```

[1]This is due to how VLAs are typically allocated on the stack, whereas `static` variables are on the heap. And the whole idea with VLAs is they'll be automatically dellocated when the stack frame is popped at the end of the function.

```
    p[3] = 34;
    printf("%d\n", v[3]);  // 34
}
```

Like with regular arrays, you can use parentheses with `sizeof()` to get the size of a would-be VLA without actually declaring one:

```
int x = 12;

printf("%zu\n", sizeof(int [x]));  // Prints 48 on my system
```

30.3 Multidimensional VLAs

You can go ahead and make all kinds of VLAs with one or more dimensions set to a variable

```
int w = 10;
int h = 20;

int x[h][w];
int y[5][w];
int z[10][w][20];
```

Again, you can navigate these just like you would a regular array.

30.4 Passing One-Dimensional VLAs to Functions

Passing single-dimensional VLAs into a function can be no different than passing a regular array in. You just go for it.

```
#include <stdio.h>

int sum(int count, int *v)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += v[i];

    return total;
}

int main(void)
{
    int x[5];   // Standard array

    int a = 5;
    int y[a];   // VLA

    for (int i = 0; i < a; i++)
        x[i] = y[i] = i + 1;

    printf("%d\n", sum(5, x));
    printf("%d\n", sum(a, y));
```

```
}
```

But there's a bit more to it than that. You can also let C know that the array is a specific VLA size by passing that in first and then giving that dimension in the parameter list:

```
int sum(int count, int v[count])
{
    // ...
}
```

Incidentally, there are a couple ways of listing a prototype for the above function; one of them involves an `*` if you don't want to specifically name the value in the VLA. It just indicates that the type is a VLA as opposed to a regular pointer.

VLA prototypes:

```
void do_something(int count, int v[count]);  // With names
void do_something(int, int v[*]);            // Without names
```

Again, that `*` thing only works with the prototype---in the function itself, you'll have to put the explicit size.

Now---*let's get multidimensional*! This is where the fun begins.

30.5 Passing Multi-Dimensional VLAs to Functions

Same thing as we did with the second form of one-dimensional VLAs, above, but this time we're passing in two dimensions and using those.

In the following example, we build a multiplication table matrix of a variable width and height, and then pass it to a function to print it out.

```
#include <stdio.h>

void print_matrix(int h, int w, int m[h][w])
{
    for (int row = 0; row < h; row++) {
        for (int col = 0; col < w; col++)
            printf("%2d ", m[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int rows = 4;
    int cols = 7;

    int matrix[rows][cols];

    for (int row = 0; row < rows; row++)
        for (int col = 0; col < cols; col++)
            matrix[row][col] = row * col;

    print_matrix(rows, cols, matrix);
}
```

30.5.1 Partial Multidimensional VLAs

You can have some of the dimensions fixed and some variable. Let's say we have a record length fixed at 5 elements, but we don't know how many records there are.

```
#include <stdio.h>

void print_records(int count, int record[count][5])
{
    for (int i = 0; i < count; i++) {
        for (int j = 0; j < 5; j++)
            printf("%2d ", record[i][j]);
        printf("\n");
    }
}

int main(void)
{
    int rec_count = 3;
    int records[rec_count][5];

    // Fill with some dummy data
    for (int i = 0; i < rec_count; i++)
        for (int j = 0; j < 5; j++)
            records[i][j] = (i+1)*(j+2);

    print_records(rec_count, records);
}
```

30.6 Compatibility with Regular Arrays

Because VLAs are just like regular arrays in memory, it's perfectly permissible to pass them interchangeably… as long as the dimensions match.

For example, if we have a function that specifically wants a 3×5 array, we can still pass a VLA into it.

```
int foo(int m[5][3]) {...}

\\ ...

int w = 3, h = 5;
int matrix[h][w];

foo(matrix);   // OK!
```

Likewise, if you have a VLA function, you can pass a regular array into it:

```
int foo(int h, int w, int m[h][w]) {...}

\\ ...

int matrix[3][5];

foo(3, 5, matrix);   // OK!
```

Beware, though: if your dimensions mismatch, you're going to have some undefined behavior going on,

likely.

30.7 `typedef` and VLAs

You can `typedef` a VLA, but the behavior might not be as you expect.

Basically, `typedef` makes a new type with the values as they existed the moment the `typedef` was executed.

So it's not a `typedef` of a VLA so much as a new fixed size array type of the dimensions at the time.

```
#include <stdio.h>

int main(void)
{
    int w = 10;

    typedef int goat[w];

    // goat is an array of 10 ints
    goat x;

    // Init with squares of numbers
    for (int i = 0; i < w; i++)
        x[i] = i*i;

    // Print them
    for (int i = 0; i < w; i++)
        printf("%d\n", x[i]);

    // Now let's change w...

    w = 20;

    // But goat is STILL an array of 10 ints, because that was the
    // value of w when the typedef executed.
}
```

So it acts like an array of fixed size.

But you still can't use an initializer list on it.

30.8 Jumping Pitfalls

You have to watch out when using `goto` near VLAs because a lot of things aren't legal.

And when you're using `longjmp()` there's a case where you could leak memory with VLAs.

But both of these things we'll cover in their respective chapters.

30.9 General Issues

VLAs have been banned from the Linux kernel for a few reasons:

- Lots of places they were used should have just been fixed-size.
- The code behind VLAs is slower (to a degree that most people wouldn't notice, but makes a difference in an operating system).

- VLAs are not supported to the same degree by all C compilers.
- Stack size is limited, and VLAs go on the stack. If some code accidentally (or maliciously) passes a large value into a kernel function that allocates a VLA, *Bad Things*™ could happen.

Other folks online point out that there's no way to detect a VLA's failure to allocate, and programs that suffered such problems would likely just crash. While fixed-size arrays also have the same issue, it's far more likely that someone accidentally make a *VLA Of Unusual Size* than somehow accidentally declare a fixed-size, say, 30 megabyte array.

Chapter 31

`goto`

The `goto` statement is universally revered and can be here presented without contest.

Just kidding! Over the years, there has been a lot of back-and-forth over whether or not (often not) `goto` is considered harmful[1].

In this programmer's opinion, you should use whichever constructs leads to the *best* code, factoring in maintainability and speed. And sometimes this might be `goto`!

In this chapter, we'll see how `goto` works in C, and then check out some of the common cases where it is used[2].

31.1 A Simple Example

In this example, we're going to use `goto` to skip a line of code and jump to a *label*. The label is the identifier that can be a `goto` target---it ends with a colon (`:`).

```
#include <stdio.h>

int main(void)
{
    printf("One\n");
    printf("Two\n");

    goto skip_3;

    printf("Three\n");

skip_3:

    printf("Five!\n");
}
```

The output is:

```
One
Two
Five!
```

[1] https://en.wikipedia.org/wiki/Goto#Criticism

[2] I'd like to point out that using `goto` in all these cases is avoidable. You can use variables and loops instead. It's just that some people think `goto` produces the *best* code in those circumstances.

`goto` sends execution jumping to the specified label, skipping everything in between.

You can jump forward or backward with `goto`.

```
infinite_loop:
    print("Hello, world!\n");
    goto infinite_loop;
```

Labels are skipped over during execution. The following will print all three numbers in order just as if the labels weren't there:

```
    printf("Zero\n");
label_1:
label_2:
    printf("One\n");
label_3:
    printf("Two\n");
label_4:
    printf("Three\n");
```

As you've noticed, it's common convention to justify the labels all the way on the left. This increases readability because a reader can quickly scan to find the destination.

Labels have *function scope*. That is, no matter how many levels deep in blocks they appear, you can still `goto` them from anywhere in the function.

It also means you can only `goto` labels that are in the same function as the `goto` itself. Labels in other functions are out of scope from `goto`'s perspective. And it means you can use the same label name in two functions---just not the same label name in the same function.

31.2 Labeled `continue`

In some languages, you can actually specify a label for a `continue` statement. C doesn't allow it, but you can easily use `goto` instead.

To show the issue, check out `continue` in this nested loop:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        continue;   // Always goes to next j
    }
}
```

As we see, that `continue`, like all `continues`, goes to the next iteration of the nearest enclosing loop. What if we want to `continue` in the next loop out, the loop with `i`?

Well, we can `break` to get back to the outer loop, right?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        break;     // Gets us to the next iteration of i
    }
}
```

That gets us two levels of nested loop. But then if we nest another loop, we're out of options. What about this, where we don't have any statement that will get us out to the next iteration of `i`?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 3; k++) {
            printf("%d, %d, %d\n", i, j, k);

            continue;  // Gets us to the next iteration of k
            break;     // Gets us to the next iteration of j
            ????;      // Gets us to the next iteration of i???

        }
    }
}
```

The `goto` statement offers us a way!

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3; k++) {
                printf("%d, %d, %d\n", i, j, k);

                goto continue_i;   // Now continuing the i loop!!
            }
        }
continue_i: ;
    }
```

We have a `;` at the end there---that's because you can't have a label pointing to the plain end of a compound statement (or before a variable declaration).

31.3 Bailing Out

When you're super nested in the middle of some code, you can use `goto` to get out of it in a manner that's often cleaner than nesting more `if`s and using flag variables.

```
    // Pseudocode

    for(...) {
        for (...) {
            while (...) {
                do {
                    if (some_error_condition)
                        goto bail;

                } while(...);
            }
        }
    }

bail:
    // Cleanup here
```

Without `goto`, you'd have to check an error condition flag in all of the loops to get all the way out.

31.4 Labeled `break`

This is a very similar situation to how `continue` only continues the innermost loop. `break` also only breaks out of the innermost loop.

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            break;   // Only breaks out of the j loop
        }
    }

    printf("Done!\n");
```

But we can use `goto` to break farther:

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            goto break_i;   // Now breaking out of the i loop!
        }
    }

break_i:

    printf("Done!\n");
```

31.5 Multi-level Cleanup

If you're calling multiple functions to initialize multiple systems and one of them fails, you should only de-initialize the ones that you've gotten to so far.

Let's do a fake example where we start initializing systems and checking to see if any returns an error (we'll use `-1` to indicate an error). If one of them does, we have to shutdown only the systems we've initialized so far.

```
    if (init_system_1() == -1)
        goto shutdown;

    if (init_system_2() == -1)
        goto shutdown_1;

    if (init_system_3() == -1)
        goto shutdown_2;

    if (init_system_4() == -1)
        goto shutdown_3;

    do_main_thing();   // Run our program

    shutdown_system4();

shutdown_3:
    shutdown_system3();
```

```
shutdown_2:
    shutdown_system2();

shutdown_1:
    shutdown_system1();

shutdown:
    print("All subsystems shut down.\n");
```

Note that we're shutting down in the reverse order that we initialized the subsystems. So if subsystem 4 fails to start up, it will shut down 3, 2, then 1 in that order.

31.6 Tail Call Optimization

Kinda. For recursive functions only.

If you're unfamiliar, Tail Call Optimization (TCO)[3] is a way to not waste stack space when calling other functions under very specific circumstances. Unfortunately the details are beyond the scope of this guide.

But if you have a recursive function you know can be optimized in this way, you can make use of this technique. (Note that you can't tail call other functions due to the function scope of labels.)

Let's do a straightforward example, factorial.

Here's a recursive version that's not TCO, but it can be!

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

int factorial(int n, int a)
{
    if (n == 0)
        return a;

    return factorial(n - 1, a * n);
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %ld\n", i, factorial(i, 1));
}
```

To make it happen, you can replace the call with two steps:

1. Set the values of the parameters to what they'd be on the next call.
2. `goto` a label on the first line of the function.

Let's try it:

```
#include <stdio.h>

int factorial(int n, int a)
{
tco:  // add this

```

[3] https://en.wikipedia.org/wiki/Tail_call

```
    if (n == 0)
        return a;

    // replace return by setting new parameter values and
    // goto-ing the beginning of the function

    //return factorial(n - 1, a * n);

    int next_n = n - 1;  // See how these match up with
    int next_a = a * n;  // the recursive arguments, above?

    n = next_n;   // Set the parameters to the new values
    a = next_a;

    goto tco;   // And repeat!
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %d\n", i, factorial(i, 1));
}
```

I used temporary variables up there to set the next values of the parameters before jumping to the start of the function. See how they correspond to the recursive arguments that were in the recursive call?

Now, why use temp variables? I could have done this instead:

```
    a *= n;
    n -= 1;

    goto tco;
```

and that actually works just fine. But if I carelessly reverse those two lines of code:

```
    n -= 1;  // BAD NEWS
    a *= n;
```

---now we're in trouble. We modified `n` before using it to modify `a`. That's Bad because that's not how it works when you call recursively. Using the temporary variables avoids this problem even if you're not looking out for it. And the compiler likely optimizes them out, anyway.

31.7 Restarting Interrupted System Calls

This is outside the spec, but commonly seen in Unix-like systems.

Certain long-lived system calls might return an error if they're interrupted by a signal, and `errno` will be set to `EINTR` to indicate the syscall was doing fine; it was just interrupted.

In those cases, it's really common for the programmer to want to restart the call and try it again.

```
retry:
    byte_count = read(0, buf, sizeof(buf) - 1);  // Unix read() syscall

    if (byte_count == -1) {            // An error occurred...
        if (errno == EINTR) {          // But it was just interrupted
            printf("Restarting...\n");
```

```
            goto retry;
        }
```

Many Unix-likes have an `SA_RESTART` flag you can pass to `sigaction()` to request the OS automatically restart any slow syscalls instead of failing with `EINTR`.

Again, this is Unix-specific and is outside the C standard.

That said, it's possible to use a similar technique any time any function should be restarted.

31.8 `goto` and Thread Preemption

This example is ripped directly from *Operating Systems: Three Easy Pieces*, another excellent book from like-minded authors who also feel that quality books should be free to download. Not that I'm opinionated, or anything.

```
retry:

    pthread_mutex_lock(L1);

    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto retry;
    }

    save_the_day();

    pthread_mutex_unlock(L2);
    pthread_mutex_unlock(L1);
```

There the thread happily acquires the mutex `L1`, but then potentially fails to get the second resource guarded by mutex `L2` (if some other uncooperative thread holds it, say). If our thread can't get the `L2` lock, it unlocks `L1` and then uses `goto` to cleanly retry.

We hope our heroic thread eventually manages to acquire both mutexes and save the day, all while avoiding evil deadlock.

31.9 `goto` and Variable Scope

We've already seen that labels have function scope, but weird things can happen if we jump past some variable initialization.

Look at this example where we jump from a place where the variable `x` is out of scope into the middle of its scope (in the block).

```
    goto label;

    {
        int x = 12345;

label:
        printf("%d\n", x);
    }
```

This will compile and run, but gives me a warning:

```
warning: ‘x’ is used uninitialized in this function
```

And then it prints out `0` when I run it (your mileage may vary).

Basically what has happened is that we jumped into `x`'s scope (so it was OK to reference it in the `printf()`) but we jumped over the line that actually initialized it to `12345`. So the value was indeterminate.

The fix is, of course, to get the initialization *after* the label one way or another.

```
    goto label;

    {
        int x;

label:
        x = 12345;
        printf("%d\n", x);
    }
```

Let's look at one more example.

```
    {
        int x = 10;

label:

        printf("%d\n", x);
    }

    goto label;
```

What happens here?

The first time through the block, we're good. `x` is `10` and that's what prints.

But after the `goto`, we're jumping into the scope of `x`, but past its initialization. Which means we can still print it, but the value is indeterminate (since it hasn't been reinitialized).

On my machine, it prints `10` again (to infinity), but that's just luck. It could print any value after the `goto` since `x` is uninitialized.

31.10 `goto` and Variable-Length Arrays

When it comes to VLAs and `goto`, there's one rule: you can't jump from outside the scope of a VLA into the scope of that VLA.

If I try to do this:

```
    int x = 10;

    goto label;

    {
        int v[x];

label:
```

```
        printf("Hi!\n");
    }
```

I get an error:

```
error: jump into scope of identifier with variably modified type
```

You can jump in ahead of the VLA declaration, like this:

```
    int x = 10;

    goto label;

    {
label:  ;
        int v[x];

        printf("Hi!\n");
    }
```

Because that way the VLA gets allocated properly before its inevitable deallocation once it falls out of scope.

Chapter 32

Types Part V: Compound Literals and Generic Selections

This is the final chapter for types! We're going to talk about two things:

- How to have ``anonymous" unnamed objects and how that's useful.
- How to generate type-dependent code.

They're not particularly related, but don't really each warrant their own chapters. So I crammed them in here like a rebel!

32.1 Compound Literals

This is a neat feature of the language that allows you to create an object of some type on the fly without ever assigning it to a variable. You can make simple types, arrays, `structs`, you name it.

One of the main uses for this is passing complex arguments to functions when you don't want to make a temporary variable to hold the value.

The way you create a compound literal is to put the type name in parentheses, and then put an initializer list after. For example, an unnamed array of `int`s, might look like this:

```
(int []){1,2,3,4}
```

Now, that line of code doesn't do anything on its own. It creates an unnamed array of 4 `int`s, and then throws them away without using them.

We could use a pointer to store a reference to the array…

```
int *p = (int []){1 ,2 ,3 ,4};

printf("%d\n", p[1]);  // 2
```

But that seems a little like a long-winded way to have an array. I mean, we could have just done this[1]:

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

printf("%d\n", p[1]);  // 2
```

So let's take a look at a more useful example.

[1] Which isn't quite the same, since it's an array, not a pointer to an `int`.

32.1.1 Passing Unnamed Objects to Functions

Let's say we have a function to sum an array of `int`s:

```
int sum(int p[], int count)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += p[i];

    return total;
}
```

If we wanted to call it, we'd normally have to do something like this, declaring an array and storing values in it to pass to the function:

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

int s = sum(a, 4);
```

But unnamed objects give us a way to skip the variable by passing it directly in (parameter names listed above). Check it out---we're going to replace the variable `a` with an unnamed array that we pass in as the first argument:

```
//                   p[]         count
//           |-----------------|  |
int s = sum((int []){1, 2, 3, 4}, 4);
```

Pretty slick!

32.1.2 Unnamed `structs`

We can do something similar with `struct`s.

First, let's do things without unnamed objects. We'll define a `struct` to hold some x/y coordinates. Then we'll define one, passing in values into its initializer. Finally, we'll pass it to a function to print the values:

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord c)
{
    printf("%d, %d\n", c.x, c.y);
}

int main(void)
{
    struct coord t = {.x=10, .y=20};

    print_coord(t);   // prints "10, 20"
}
```

Straightforward enough?

Let's modify it to use an unnamed object instead of the variable `t` we're passing to `print_coord()`.

We'll just take `t` out of there and replace it with an unnamed `struct`:

```
    //struct coord t = {.x=10, .y=20};

    print_coord((struct coord){.x=10, .y=20});   // prints "10, 20"
```

Still works!

32.1.3 Pointers to Unnamed Objects

You might have noticed in the last example that even through we were using a `struct`, we were passing a copy of the `struct` to `print_coord()` as opposed to passing a pointer to the `struct`.

Turns out, we can just take the address of an unnamed object with `&` like always.

This is because, in general, if an operator would have worked on a variable of that type, you can use that operator on an unnamed object of that type.

Let's modify the above code so that we pass a pointer to an unnamed object

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord *c)
{
    printf("%d, %d\n", c->x, c->y);
}

int main(void)
{
    //     Note the &
    //           |
    print_coord(&(struct coord){.x=10, .y=20});   // prints "10, 20"
}
```

Additionally, this can be a nice way to pass even pointers to simple objects:

```
// Pass a pointer to an int with value 3490
foo(&(int){3490});
```

Easy as that.

32.1.4 Unnamed Objects and Scope

The lifetime of an unnamed object ends at the end of its scope. The biggest way this could bite you is if you make a new unnamed object, get a pointer to it, and then leave the object's scope. In that case, the pointer will refer to a dead object.

So this is undefined behavior:

```
int *p;

{
    p = &(int){10};
}
```

```
printf("%d\n", *p);  // INVALID: The (int){10} fell out of scope
```

Likewise, you can't return a pointer to an unnamed object from a function. The object is deallocated when it falls out of scope:

```
#include <stdio.h>

int *get3490(void)
{
    // Don't do this
    return &(int){3490};
}

int main(void)
{
    printf("%d\n", *get3490());  // INVALID: (int){3490} fell out of scope
}
```

Just think of their scope like that of an ordinary local variable. You can't return a pointer to a local variable, either.

32.1.5 Silly Unnamed Object Example

You can put any type in there and make an unnamed object.

For example, these are effectively equivalent:

```
int x = 3490;

printf("%d\n", x);              // 3490 (variable)
printf("%d\n", 3490);           // 3490 (constant)
printf("%d\n", (int){3490});    // 3490 (unnamed object)
```

That last one is unnamed, but it's silly. Might as well do the simple one on the line before.

But hopefully that provides a little more clarity on the syntax.

32.2 Generic Selections

This is an expression that allows you to select different pieces of code depending on the *type* of the first argument to the expression.

We'll look at an example in just a second, but it's important to know this is processed at compile time, *not at runtime*. There's no runtime analysis going on here.

The expression begins with `_Generic`, works kinda like a `switch`, and it takes at least two arguments.

The first argument is an expression (or variable[2]) that has a *type*. All expressions have a type. The remaining arguments to `_Generic` are the cases of what to substitute in for the result of the expression if the first argument is that type.

Wat?

Let's try it out and see.

[2] A variable used here *is* an expression.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i;
6      float f;
7      char c;
8
9      char *s = _Generic(i,
10                        int: "that variable is an int",
11                        float: "that variable is a float",
12                        default: "that variable is some type"
13                    );
14
15     printf("%s\n", s);
16 }
```

Check out the `_Generic` expression starting on line 9.

When the compiler sees it, it looks at the type of the first argument. (In this example, the type of the variable `i`.) It then looks through the cases for something of that type. And then it substitutes the argument in place of the entire `_Generic` expression.

In this case, `i` is an `int`, so it matches that case. Then the string is substituted in for the expression. So the line turns into this when the compiler sees it:

```
    char *s = "that variable is an int";
```

If the compiler can't find a type match in the `_Generic`, it looks for the optional `default` case and uses that.

If it can't find a type match and there's no `default`, you'll get a compile error. The first expression **must** match one of the types or `default`.

Because it's inconvenient to write `_Generic` over and over, it's often used to make the body of a macro that can be easily repeatedly reused.

Let's make a macro `TYPESTR(x)` that takes an argument and returns a string with the type of the argument.

So `TYPESTR(1)` will return the string `"int"`, for example.

Here we go:

```
#include <stdio.h>

#define TYPESTR(x) _Generic((x), \
                        int: "int", \
                        long: "long", \
                        float: "float", \
                        double: "double", \
                        default: "something else")

int main(void)
{
    int i;
    long l;
    float f;
    double d;
    char c;
```

```
    printf("i is type %s\n", TYPESTR(i));
    printf("l is type %s\n", TYPESTR(l));
    printf("f is type %s\n", TYPESTR(f));
    printf("d is type %s\n", TYPESTR(d));
    printf("c is type %s\n", TYPESTR(c));
}
```

This outputs:

```
i is type int
l is type long
f is type float
d is type double
c is type something else
```

Which should be no surprise, because, like we said, that code in `main()` is replaced with the following when it is compiled:

```
    printf("i is type %s\n", "int");
    printf("l is type %s\n", "long");
    printf("f is type %s\n", "float");
    printf("d is type %s\n", "double");
    printf("c is type %s\n", "something else");
```

And that's exactly the output we see.

Let's do one more. I've included some macros here so that when you run:

```
int i = 10;
char *s = "Foo!";

PRINT_VAL(i);
PRINT_VAL(s);
```

you get the output:

```
i = 10
s = Foo!
```

We'll have to make use of some macro magic to do that.

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

// Macro that gives back a format specifier for a type
#define FMTSPEC(x) _Generic((x), \
                        int: "%d", \
                        long: "%ld", \
                        float: "%f", \
                        double: "%f", \
                        char *: "%s")
                        // TODO: add more types

// Macro that prints a variable in the form "name = value"
#define PRINT_VAL(x) do { \
    char fmt[512]; \
    snprintf(fmt, sizeof fmt, #x " = %s\n", FMTSPEC(x)); \
    printf(fmt, (x)); \
```

```
} while(0)

int main(void)
{
    int i = 10;
    float f = 3.14159;
    char *s = "Hello, world!";

    PRINT_VAL(i);
    PRINT_VAL(f);
    PRINT_VAL(s);
}
```

for the output:

```
i = 10
f = 3.141590
s = Hello, world!
```

We could have crammed that all in one big macro, but I broke it into two to prevent eye bleeding.

Chapter 33

Arrays Part II

We're going to go over a few extra misc things this chapter concerning arrays.

- Type qualifiers with array parameters
- The `static` keyword with array parameters
- Partial multi-dimensional array initializers

They're not super-commonly seen, but we'll peek at them since they're part of the newer spec.

33.1 Type Qualifiers for Arrays in Parameter Lists

If you recall from earlier, these two things are equivalent in function parameter lists:

```
int func(int *p) {...}
int func(int p[]) {...}
```

And you might also recall that you can add type qualifiers to a pointer variable like so:

```
int *const p;
int *volatile p;
int *const volatile p;
// etc.
```

But how can we do that when we're using array notation in your parameter list?

Turns out it goes in the brackets. And you can put the optional count after. The two following lines are equivalent:

```
int func(int *const volatile p) {...}
int func(int p[const volatile]) {...}
int func(int p[const volatile 10]) {...}
```

If you have a multidimensional array, you need to put the type qualifiers in the first set of brackets.

33.2 `static` for Arrays in Parameter Lists

Similarly, you can use the keyword static in the array in a parameter list.

This is something I've never seen in the wild. It is **always** followed by a dimension:

```
int func(int p[static 4]) {...}
```

What this means, in the above example, is the compiler is going to assume that any array you pass to the function will be *at least* 4 elements.

Anything else is undefined behavior.

```
int func(int p[static 4]) {...}

int main(void)
{
    int a[] = {11, 22, 33, 44};
    int b[] = {11, 22, 33, 44, 55};
    int c[] = {11, 22};

    func(a);  // OK! a is 4 elements, the minimum
    func(b);  // OK! b is at least 4 elements
    func(c);  // Undefined behavior! c is under 4 elements!
}
```

This basically sets the minimum size array you can have.

Important note: there is nothing in the compiler that prohibits you from passing in a smaller array. The compiler probably won't warn you, and it won't detect it at runtime.

By putting `static` in there, you're saying, ``I double secret PROMISE that I will never pass in a smaller array than this." And the compiler says, ``Yeah, fine," and trusts you to not do it.

And then the compiler can make certain code optimizations, safe in the knowledge that you, the programmer, will always do the right thing.

33.3 Equivalent Initializers

C is a little bit, shall we say, *flexible* when it comes to array initializers.

We've already seen some of this, where any missing values are replaced with zero.

For example, we can initialize a 5 element array to `1,2,0,0,0` with this:

```
int a[5] = {1, 2};
```

Or set an array entirely to zero with:

```
int a[5] = {0};
```

But things get interesting when initializing multidimensional arrays.

Let's make an array of 3 rows and 2 columns:

```
int a[3][2];
```

Let's write some code to initialize it and print the result:

```
#include <stdio.h>

int main(void)
{
    int a[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };
```

```
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 2; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}
```

And when we run it, we get the expected:

```
1 2
3 4
5 6
```

Let's leave off some of the initializer elements and see they get set to zero:

```
    int a[3][2] = {
        {1, 2},
        {3},    // Left off the 4!
        {5, 6}
    };
```

which produces:

```
1 2
3 0
5 6
```

Now let's leave off the entire last middle element:

```
    int a[3][2] = {
        {1, 2},
        // {3, 4},   // Just cut this whole thing out
        {5, 6}
    };
```

And now we get this, which might not be what you expect:

```
1 2
5 6
0 0
```

But if you stop to think about it, we only provided enough initializers for two rows, so they got used for the first two rows. And the remaining elements were initialized to zero.

So far so good. Generally, if we leave off parts of the initializer, the compiler sets the corresponding elements to `0`.

But let's get *crazy*.

```
    int a[3][2] = { 1, 2, 3, 4, 5, 6 };
```

What---? That's a 2D array, but it only has a 1D initializer!

Turns out that's legal (though GCC will warn about it with the proper warnings turned on).

Basically, what it does is starts filling in elements in row 0, then row 1, then row 2 from left to right.

So when we print, it prints in order:

```
1 2
3 4
5 6
```

If we leave some off:

```
    int a[3][2] = { 1, 2, 3 };
```

they fill with `0`:

```
1 2
3 0
0 0
```

So if you want to fill the whole array with `0`, then go ahead and:

```
    int a[3][2] = {0};
```

But my recommendation is if you have a 2D array, use a 2D initializer. It just makes the code more readable. (Except for initializing the whole array with `0`, in which case it's idiomatic to use `{0}` no matter the dimension of the array.)

Chapter 34

Long Jumps with `setjmp`, `longjmp`

We've already seen `goto`, which jumps in function scope. But `longjmp()` allows you to jump back to an earlier point in execution, back to a function that called this one.

There are a lot of limitations and caveats, but this can be a useful function for bailing out from deep in the call stack back up to an earlier state.

In my experience, this is very rarely-used functionality.

34.1 Using `setjmp` and `longjmp`

The dance we're going to do here is to basically put a bookmark in execution with `setjmp()`. Later on, we'll call `longjmp()` and it'll jump back to the earlier point in execution where we set the bookmark with `setjmp()`.

And it can do this even if you've called subfunctions.

Here's a quick demo where we call into functions a couple levels deep and then bail out of it.

We're going to use a file scope variable `env` to keep the *state* of things when we call `setjmp()` so we can restore them when we call `longjmp()` later. This is the variable in which we remember our ``place''.

The variable `env` is of type `jmp_buf`, an opaque type declared in `<setjmp.h>`.

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

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Bail out
    printf("Leaving depth 2\n");  // This won't happen
}

void depth1(void)
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // This won't happen
```

```
}

int main(void)
{
    switch (setjmp(env)) {
      case 0:
          printf("Calling into functions, setjmp() returned 0\n");
          depth1();
          printf("Returned from functions\n");  // This won't happen
          break;

      case 3490:
          printf("Bailed back to main, setjmp() returned 3490\n");
          break;
    }
}
```

When run, this outputs:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

If you try to take that output and match it up with the code, it's clear there's some really *funky* stuff going on.

One of the most notable things is that `setjmp()` returns *twice*. What the actual frank? What is this sorcery?!

So here's the deal: if `setjmp()` returns `0`, it means that you've successfully set the ``bookmark'' at that point.

If it returns non-zero, it means you've just returned to the ``bookmark'' set earlier. (And the value returned is the one you pass to `longjmp()`.)

This way you can tell the difference between setting the bookmark and returning to it later.

So when the code, above, calls `setjmp()` the first time, `setjmp()` *stores* the state in the `env` variable and returns `0`. Later when we call `longjmp()` with that same `env`, it restores the state and `setjmp()` returns the value `longjmp()` was passed.

34.2 Pitfalls

Under the hood, this is pretty straightforward. Typically the *stack pointer* keeps track of the locations in memory that local variables are stored, and the *program counter* keeps track of the address of the currently-executing instruction[1].

So if we want to jump back to an earlier function, it's basically only a matter of restoring the stack pointer and program counter to the values kept in the `jmp_buf` variable, and making sure the return value is set correctly. And then execution will resume there.

But a variety of factors confound this, making a significant number of undefined behavior traps.

34.2.1 The Values of Local Variables

If you want the values of automatic (non-`static` and non-`extern`) local variables to persist in the function that called `setjmp()` after a `longjmp()` happens, you must declare those variables to be `volatile`.

[1] Both ``stack pointer'' and ``program counter'' are related to the underlying architecture and C implementation, and are not part of the spec.

Technically, they only have to be `volatile` if they change between the time `setjmp()` is called and `longjmp()` is called[2].

For example, if we run this code:

```
int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

and then later `longjmp()` back, the value of `x` will be indeterminate.

If we want to fix this, `x` must be `volatile`:

```
volatile int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

Now the value will be the correct `30` after a `longjmp()` returns us to this point.

34.2.2 How Much State is Saved?

When you `longjmp()`, execution resumes at the point of the corresponding `setjmp()`. And that's it.

The spec points out that it's just as if you'd jumped back into the function at that point with local variables set to whatever values they had when the `longjmp()` call was made.

Things that aren't restored include, paraphrasing the spec:

- Floating point status flags
- Open files
- Any other component of the abstract machine

34.2.3 You Can't Name Anything `setjmp`

You can't have any `extern` identifiers with the name `setjmp`. Or, if `setjmp` is a macro, you can't undefine it.

Both are undefined behavior.

34.2.4 You Can't `setjmp()` in a Larger Expression

That is, you can't do something like this:

```
if (x == 12 && setjmp(env) == 0) { ... }
```

That's too complex to be allowed by the spec due to the machinations that must occur when unrolling the stack and all that. We can't `longjmp()` back into some complex expression that's only been partially executed.

So there are limits on the complexity of that expression.

- It can be the entire controlling expression of the conditional.

  ```
  if (setjmp(env)) {...}
  ```

[2]The rationale here is that the program might store a value temporarily in a *CPU register* while it's doing work on it. In that timeframe, the register holds the correct value, and the value on the stack might be out of date. Then later the register values would get overwritten and the changes to the variable lost.

```
switch (setjmp(env)) {...}
```

- It can be part of a relational or equality expression, as long as the other operand is an integer constant. And the whole thing is the controlling expression of the conditional.

```
if (setjmp(env) == 0) {...}
```

- The operand to a logical NOT (`!`) operation, being the entire controlling expression.

```
if (!setjmp(env)) {...}
```

- A standalone expression, possibly cast to `void`.

```
setjmp(env);
```

```
(void)setjmp(env);
```

34.2.5 When Can't You `longjmp()`?

It's undefined behavior if:

- You didn't call `setjmp()` earlier
- You called `setjmp()` from another thread
- You called `setjmp()` in the scope of a variable length array (VLA), and execution left the scope of that VLA before `longjmp()` was called.
- The function containing the `setjmp()` exited before `longjmp()` was called.

On that last one, ``exited'' includes normal returns from the function, as well as the case if another `longjmp()` jumped back to ``earlier'' in the call stack than the function in question.

34.2.6 You Can't Pass `0` to `longjmp()`

If you try to pass the value `0` to `longjmp()`, it will silently change that value to `1`.

Since `setjmp()` ultimately returns this value, and having `setjmp()` return `0` has special meaning, returning `0` is prohibited.

34.2.7 `longjmp()` and Variable Length Arrays

If you are in scope of a VLA and `longjmp()` out there, the memory allocated to the VLA could leak[3].

Same thing happens if you `longjmp()` back over any earlier functions that had VLAs still in scope.

This is one thing that really bugged me able VLAs---that you could write perfectly legitimate C code that squandered memory. But, hey---I'm not in charge of the spec.

[3]That is, remain allocated until the program ends with no way to free it.

Chapter 35

Incomplete Types

It might surprise you to learn that this builds without error:

```
extern int a[];

int main(void)
{
    struct foo *x;
    union bar *y;
    enum baz *z;
}
```

We never gave a size for `a`. And we have pointers to `structs` `foo`, `bar`, and `baz` that never seem to be declared anywhere.

And the only warnings I get are that `x`, `y`, and `z` are unused.

These are examples of *incomplete types*.

An incomplete type is a type the size (i.e. the size you'd get back from `sizeof`) for which is not known. Another way to think of it is a type that you haven't finished declaring.

You can have a pointer to an incomplete type, but you can't dereference it or use pointer arithmetic on it. And you can't `sizeof` it.

So what can you do with it?

35.1 Use Case: Self-Referential Structures

I only know of one real use case: forward references to `structs` or `unions` with self-referential or co-dependent structures. (I'm going to use `struct` for the rest of these examples, but they all apply equally to `unions`, as well.)

Let's do the classic example first.

But before I do, know this! As you declare a `struct`, the `struct` is incomplete until the closing brace is reached!

```
struct antelope {              // struct antelope is incomplete here
    int leg_count;             // Still incomplete
    float stomach_fullness;    // Still incomplete
    float top_speed;           // Still incomplete
```

```
    char *nickname;            // Still incomplete
};                             // NOW it's complete.
```

So what? Seems sane enough.

But what if we're doing a linked list? Each linked list node needs to have a reference to another node. But how can we create a reference to another node if we haven't finished even declaring the node yet?

C's allowance for incomplete types makes it possible. We can't declare a node, but we *can* declare a pointer to one, even if it's incomplete!

```
struct node {
    int val;
    struct node *next;  // struct node is incomplete, but that's OK!
};
```

Even though the `struct node` is incomplete on line 3, we can still declare a pointer to one[1].

We can do the same thing if we have two different `structs` that refer to each other:

```
struct a {
    struct b *x;  // Refers to a `struct b`
};

struct b {
    struct a *x;  // Refers to a `struct a`
};
```

We'd never be able to make that pair of structures without the relaxed rules for incomplete types.

35.2 Incomplete Type Error Messages

Are you getting errors like these?

```
invalid application of ‘sizeof’ to incomplete type

invalid use of undefined type

dereferencing pointer to incomplete type
```

Most likely culprit: you probably forgot to `#include` the header file that declares the type.

35.3 Other Incomplete Types

Declaring a `struct` or `union` with no body makes an incomplete type, e.g. `struct foo;`.

`enums` are incomplete until the closing brace.

`void` is an incomplete type.

Arrays declared `extern` with no size are incomplete, e.g.:

```
extern int a[];
```

If it's a non-`extern` array with no size followed by an initializer, it's incomplete until the closing brace of the initializer.

[1]This works because in C, pointers are the same size regardless of the type of data they point to. So the compiler doesn't need to know the size of the `struct node` at this point; it just needs to know the size of a pointer.

35.4 Use Case: Arrays in Header Files

It can be useful to declare incomplete array types in header files. In those cases, the actual storage (where the complete array is declared) should be in a single `.c` file. If you put it in the `.h` file, it will be duplicated every time the header file is included.

So what you can do is make a header file with an incomplete type that refers to the array, like so:

```
// File: bar.h

#ifndef BAR_H
#define BAR_H

extern int my_array[];  // Incomplete type

#endif
```

And the in the `.c` file, actually define the array:

```
// File: bar.c

int my_array[1024];     // Complete type!
```

Then you can include the header from as many places as you'd like, and every one of those places will refer to the same underlying `my_array`.

```
// File: foo.c

#include <stdio.h>
#include "bar.h"    // includes the incomplete type for my_array

int main(void)
{
    my_array[0] = 10;

    printf("%d\n", my_array[0]);
}
```

When compiling multiple files, remember to specify all the `.c` files to the compiler, but not the `.h` files, e.g.:

```
gcc -o foo foo.c bar.c
```

35.5 Completing Incomplete Types

If you have an incomplete type, you can complete it by defining the complete `struct`, `union`, `enum`, or array in the same scope.

```
struct foo;         // incomplete type

struct foo *p;      // pointer, no problem

// struct foo f;    // Error: incomplete type!

struct foo {
    int x, y, z;
};                  // Now the struct foo is complete!
```

```
struct foo f;      // Success!
```

Note that though `void` is an incomplete type, there's no way to complete it. Not that anyone ever thinks of doing that weird thing. But it does explain why you can do this:

```
void *p;             // OK: pointer to incomplete type
```

and not either of these:

```
void v;              // Error: declare variable of incomplete type

printf("%d\n", *p);  // Error: dereference incomplete type
```

The more you know…

Chapter 36

Complex Numbers

A tiny primer on Complex numbers[1] stolen directly from Wikipedia:

> A **complex number** is a number that can be expressed in the form $a + bi$, where a and b are real numbers [i.e. floating point types in C], and i represents the imaginary unit, satisfying the equation $i^2 = -1$. Because no real number satisfies this equation, i is called an imaginary number. For the complex number $a + bi$, a is called the **real part**, and b is called the **imaginary part**.

But that's as far as I'm going to go. We'll assume that if you're reading this chapter, you know what a complex number is and what you want to do with them.

And all we need to cover is C's faculties for doing so.

Turns out, though, that complex number support in a compiler is an *optional* feature. Not all compliant compilers can do it. And the ones that do, might do it to various degrees of completeness.

You can test if your system supports complex numbers with:

```
#ifdef __STDC_NO_COMPLEX__
#error Complex numbers not supported!
#endif
```

Furthermore, there is a macro that indicates adherence to the ISO 60559 (IEEE 754) standard for floating point math with complex numbers, as well as the presence of the `_Imaginary` type.

```
#if __STDC_IEC_559_COMPLEX__ != 1
#error Need IEC 60559 complex support!
#endif
```

More details on that are spelled out in Annex G in the C11 spec.

36.1 Complex Types

To use complex numbers, `#include <complex.h>`.

With that, you get at least two types:

```
_Complex
complex
```

Those both mean the same thing, so you might as well use the prettier `complex`.

[1] https://en.wikipedia.org/wiki/Complex_number

You also get some types for imaginary numbers if you implementation is IEC 60559-compliant:

```
_Imaginary
imaginary
```

These also both mean the same thing, so you might as well use the prettier `imaginary`.

You also get values for the imaginary number i, itself:

```
I
_Complex_I
_Imaginary_I
```

The macro `I` is set to `_Imaginary_I` (if available), or `_Complex_I`. So just use `I` for the imaginary number.

One aside: I've said that if a compiler has `__STDC_IEC_559_COMPLEX__` set to `1`, it must support `_Imaginary` types to be compliant. That's my read of the spec. However, I don't know of a single compiler that actually supports `_Imaginary` even though they have `__STDC_IEC_559_COMPLEX__` set. So I'm going to write some code with that type in here I have no way of testing. Sorry!

OK, so now we know there's a `complex` type, how can we use it?

36.2 Assigning Complex Numbers

Since the complex number has a real and imaginary part, but both of them rely on floating point numbers to store values, we need to also tell C what precision to use for those parts of the complex number.

We do that by just pinning a `float`, `double`, or `long double` to the `complex`, either before or after it.

Let's define a complex number that uses `float` for its components:

```
float complex c;   // Spec prefers this way
complex float c;   // Same thing--order doesn't matter
```

So that's great for declarations, but how do we initialize them or assign to them?

Turns out we get to use some pretty natural notation. Example!

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;
```

For $5 + 2i$ and $10 + 3i$, respectively.

36.3 Constructing, Deconstructing, and Printing

We're getting there…

We've already seen one way to write a complex number:

```
double complex x = 5 + 2*I;
```

There's also no problem using other floating point numbers to build it:

```
double a = 5;
double b = 2;
double complex x = a + b*I;
```

There is also a set of macros to help build these. The above code could be written using the `CMPLX()` macro, like so:

```
double complex x = CMPLX(5, 2);
```

As far as I can tell in my research, these are *almost* equivalent:

```
double complex x = 5 + 2*I;
double complex x = CMPLX(5, 2);
```

But the `CMPLX()` macro will handle negative zeros in the imaginary part correctly every time, whereas the other way might convert them to positive zeros. I *think*[2] This seems to imply that if there's a chance the imaginary part will be zero, you should use the macro… but someone should correct me on this if I'm mistaken!

The `CMPLX()` macro works on `double` types. There are two other macros for `float` and `long double`: `CMPLXF()` and `CMPLXL()`. (These ``f'' and ``l'' suffixes appear in virtually all the complex-number-related functions.)

Now let's try the reverse: if we have a complex number, how do we break it apart into its real and imaginary parts?

Here we have a couple functions that will extract the real and imaginary parts from the number: `creal()` and `cimag()`:

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;

printf("x = %f + %fi\n", creal(x), cimag(x));
printf("y = %f + %fi\n", creal(y), cimag(y));
```

for the output:

```
x = 5.000000 + 2.000000i
y = 10.000000 + 3.000000i
```

Note that the `i` I have in the `printf()` format string is a literal `i` that gets printed---it's not part of the format specifier. Both return values from `creal()` and `cimag()` are `double`.

And as usual, there are `float` and `long double` variants of these functions: `crealf()`, `cimagf()`, `creall()`, and `cimagl()`.

36.4 Complex Arithmetic and Comparisons

Arithmetic can be performed on complex numbers, though how this works mathematically is beyond the scope of the guide.

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

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;
    double complex z;

    z = x + y;
    printf("x + y = %f + %fi\n", creal(z), cimag(z));
```

[2]This was a harder one to research, and I'll take any more information anyone can give me. `I` could be defined as `_Complex_I` or `_Imaginary_I`, if the latter exists. `_Imaginary_I` will handle signed zeros, but `_Complex_I` *might* not. This has implications with branch cuts and other complex-numbery-mathy things. Maybe. Can you tell I'm really getting out of my element here? In any case, the `CMPLX()` macros behave as if `I` were defined as `_Imaginary_I`, with signed zeros, even if `_Imaginary_I` doesn't exist on the system.

```
    z = x - y;
    printf("x - y = %f + %fi\n", creal(z), cimag(z));

    z = x * y;
    printf("x * y = %f + %fi\n", creal(z), cimag(z));

    z = x / y;
    printf("x / y = %f + %fi\n", creal(z), cimag(z));
}
```

for a result of:

```
x + y = 4.000000 + 6.000000i
x - y = -2.000000 + -2.000000i
x * y = -5.000000 + 10.000000i
x / y = 0.440000 + 0.080000i
```

You can also compare two complex numbers for equality (or inequality):

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

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;

    printf("x == y = %d\n", x == y);  // 0
    printf("x != y = %d\n", x != y);  // 1
}
```

with the output:

```
x == y = 0
x != y = 1
```

They are equal if both components test equal. Note that as with all floating point, they could be equal if they're close enough due to rounding error[3].

36.5 Complex Math

But wait! There's more than just simple complex arithmetic!

Here's a summary table of all the math functions available to you with complex numbers.

I'm only going to list the `double` version of each function, but for all of them there is a `float` version that you can get by appending `f` to the function name, and a `long double` version that you can get by appending `l`.

For example, the `cabs()` function for computing the absolute value of a complex number also has `cabsf()` and `cabsl()` variants. I'm omitting them for brevity.

36.5.1 Trigonometry Functions

[3]The simplicity of this statement doesn't do justice to the incredible amount of work that goes into simply understanding how floating point actually functions. https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

Function	Description
`ccos()`	Cosine
`csin()`	Sine
`ctan()`	Tangent
`cacos()`	Arc cosine
`casin()`	Arc sine
`catan()`	Play *Settlers of Catan*
`ccosh()`	Hyperbolic cosine
`csinh()`	Hyperbolic sine
`ctanh()`	Hyperbolic tangent
`cacosh()`	Arc hyperbolic cosine
`casinh()`	Arc hyperbolic sine
`catanh()`	Arc hyperbolic tangent

36.5.2 Exponential and Logarithmic Functions

Function	Description
`cexp()`	Base-e exponential
`clog()`	Natural (base-e) logarithm

36.5.3 Power and Absolute Value Functions

Function	Description
`cabs()`	Absolute value
`cpow()`	Power
`csqrt()`	Square root

36.5.4 Manipulation Functions

Function	Description
`creal()`	Return real part
`cimag()`	Return imaginary part
`CMPLX()`	Construct a complex number
`carg()`	Argument/phase angle
`conj()`	Conjugate[4]
`cproj()`	Projection on Riemann sphere

[4]This is the only one that doesn't begin with an extra leading `c`, strangely.

Chapter 37

Fixed Width Integer Types

C has all those small, bigger, and biggest integer types like `int` and `long` and all that. And you can look in the section on limits to see what the largest int is with `INT_MAX` and so on.

How big are those types? That is, how many bytes do they take up? We could use `sizeof` to get that answer.

But what if I wanted to go the other way? What if I needed a type that was exactly 32 bits (4 bytes) or at least 16 bits or somesuch?

How can we declare a type that's a certain size?

The header `<stdint.h>` gives us a way.

37.1 The Bit-Sized Types

For both signed and unsigned integers, we can specify a type that is a certain number of bits, with some caveats, of course.

And there are three main classes of these types (in these examples, the `N` would be replaced by a certain number of bits):

- Integers of exactly a certain size (`intN_t`)
- Integers that are at least a certain size (`int_leastN_t`)
- Integers that are at least a certain size and are as fast as possible (`int_fastN_t`)[1]

How much faster is `fast`? Definitely maybe some amount faster. Probably. The spec doesn't say how much faster, just that they'll be the fastest on this architecture. Most C compilers are pretty good, though, so you'll probably only see this used in places where the most possible speed needs to be guaranteed (rather than just hoping the compiler is producing pretty-dang-fast code, which it is).

Finally, these unsigned number types have a leading `u` to differentiate them.

For example, these types have the corresponding listed meaning:

```
int32_t w;        // x is exactly 32 bits, signed
uint16_t x;       // y is exactly 16 bits, unsigned

int_least8_t y;   // y is at least 8 bits, signed

uint_fast64_t z;  // z is the fastest representation at least 64 bits, unsigned
```

[1] Some architectures have different sized data that the CPU and RAM can operate with at a faster rate than others. In those cases, if you need the fastest 8-bit number, it might give you have a 16- or 32-bit type instead because that's just faster. So with this, you won't know how big the type is, but it will be least as big as you say.

The following types are guaranteed to be defined:

```
int_least8_t      uint_least8_t
int_least16_t     uint_least16_t
int_least32_t     uint_least32_t
int_least64_t     uint_least64_t

int_fast8_t       uint_fast8_t
int_fast16_t      uint_fast16_t
int_fast32_t      uint_fast32_t
int_fast64_t      uint_fast64_t
```

There might be others of different widths, as well, but those are optional.

Hey! Where are the fixed types like `int16_t`? Turns out those are entirely optional…unless certain conditions are met[2]. And if you have an average run-of-the-mill modern computer system, those conditions probably are met. And if they are, you'll have these types:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

Other variants with different widths might be defined, but they're optional.

37.2 Maximum Integer Size Type

There's a type you can use that holds the largest representable integers available on the system, both signed and unsigned:

```
intmax_t
uintmax_t
```

Use these types when you want to go as big as possible.

Obviously values from any other integer types of the same sign will fit in this type, necessarily.

37.3 Using Fixed Size Constants

If you have a constant that you want to have fit in a certain number of bits, you can use these macros to automatically append the proper suffix onto the number (e.g. `22L` or `3490ULL`).

```
INT8_C(x)     UINT8_C(x)
INT16_C(x)    UINT16_C(x)
INT32_C(x)    UINT32_C(x)
INT64_C(x)    UINT64_C(x)
INTMAX_C(x)   UINTMAX_C(x)
```

Again, these work only with constant integer values.

For example, we can use one of these to assign constant values like so:

```
uint16_t x = UINT16_C(12);
intmax_t y = INTMAX_C(3490);
```

[2]Namely, the system has 8, 16, 32, or 64 bit integers with no padding that use two's complement representation, in which case the `intN_t` variant for that particular number of bits *must* be defined.

37.4 Limits of Fixed Size Integers

We also have some limits defined so you can get the maximum and minimum values for these types:

```
INT8_MAX           INT8_MIN           UINT8_MAX
INT16_MAX          INT16_MIN          UINT16_MAX
INT32_MAX          INT32_MIN          UINT32_MAX
INT64_MAX          INT64_MIN          UINT64_MAX

INT_LEAST8_MAX     INT_LEAST8_MIN     UINT_LEAST8_MAX
INT_LEAST16_MAX    INT_LEAST16_MIN    UINT_LEAST16_MAX
INT_LEAST32_MAX    INT_LEAST32_MIN    UINT_LEAST32_MAX
INT_LEAST64_MAX    INT_LEAST64_MIN    UINT_LEAST64_MAX

INT_FAST8_MAX      INT_FAST8_MIN      UINT_FAST8_MAX
INT_FAST16_MAX     INT_FAST16_MIN     UINT_FAST16_MAX
INT_FAST32_MAX     INT_FAST32_MIN     UINT_FAST32_MAX
INT_FAST64_MAX     INT_FAST64_MIN     UINT_FAST64_MAX

INTMAX_MAX         INTMAX_MIN         UINTMAX_MAX
```

Note the `MIN` for all the unsigned types is `0`, so, as such, there's no macro for it.

37.5 Format Specifiers

In order to print these types, you need to send the right format specifier to `printf()`. (And the same issue for getting input with `scanf()`.)

But how are you going to know what size the types are under the hood? Luckily, once again, C provides some macros to help with this.

All this can be found in `<inttypes.h>`.

Now, we have a bunch of macros. Like a complexity explosion of macros. So I'm going to stop listing out every one and just put the lowercase letter `n` in the place where you should put `8`, `16`, `32`, or `64` depending on your needs.

Let's look at the macros for printing signed integers:

```
PRIdn    PRIdLEASTn    PRIdFASTn    PRIdMAX
PRIin    PRIiLEASTn    PRIiFASTn    PRIiMAX
```

Look for the patterns there. You can see there are variants for the fixed, least, fast, and max types.

And you also have a lowercase `d` and a lowercase `i`. Those correspond to the `printf()` format specifiers `%d` and `%i`.

So if I have something of type:

```
int_least16_t x = 3490;
```

I can print that with the equivalent format specifier for `%d` by using `PRIdLEAST16`.

But how? How do we use that macro?

First of all, that macro specifies a string containing the letter or letters `printf()` needs to use to print that type. Like, for example, it could be `"d"` or `"ld"`.

So all we need to do is embed that in our format string to the `printf()` call.

To do this, we can take advantage of a fact about C that you might have forgotten: adjacent string literals are automatically concatenated to a single string. E.g.:

```
printf("Hello, " "world!\n");   // Prints "Hello, world!"
```

And since these macros are string literals, we can use them like so:

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

int main(void)
{
    int_least16_t x = 3490;

    printf("The value is %" PRIdLEAST16 "!\n", x);
}
```

We also have a pile of macros for printing unsigned types:

```
PRIon    PRIoLEASTn    PRIoFASTn    PRIoMAX
PRIun    PRIuLEASTn    PRIuFASTn    PRIuMAX
PRIxn    PRIxLEASTn    PRIxFASTn    PRIxMAX
PRIXn    PRIXLEASTn    PRIXFASTn    PRIXMAX
```

In this case, `o`, `u`, `x`, and `X` correspond to the documented format specifiers in `printf()`.

And, as before, the lowercase `n` should be substituted with `8`, `16`, `32`, or `64`.

But just when you think you had enough of the macros, it turns out we have a complete complementary set of them for `scanf()`!

```
SCNdn    SCNdLEASTn    SCNdFASTn    SCNdMAX
SCNin    SCNiLEASTn    SCNiFASTn    SCNiMAX
SCNon    SCNoLEASTn    SCNoFASTn    SCNoMAX
SCNun    SCNuLEASTn    SCNuFASTn    SCNuMAX
SCNxn    SCNxLEASTn    SCNxFASTn    SCNxMAX
```

Remember: when you want to print out a fixed size integer type with `printf()` or `scanf()`, grab the correct corresponding format specifer from `<inttypes.h>`.

Chapter 38

Date and Time Functionality

> ``Time is an illusion. Lunchtime doubly so."
> ---Ford Prefect, The Hitchhikers Guide to the Galaxy

This isn't too complex, but it can be a little intimidating at first, both with the different types available and the way we can convert between them.

Mix in GMT (UTC) and local time and we have all the *Usual Fun*™ one gets with times and dates.

And of course never forget the golden rule of dates and times: *Never attempt to write your own date and time functionality. Only use what the library gives you.*

Time is too complex for mere mortal programmers to handle correctly. Seriously, we all owe a point to everyone who worked on any date and time library, so put that in your budget.

38.1 Quick Terminology and Information

Just a couple quick terms in case you don't have them down.

- **UTC**: Coordinated Universal Time is a universally[1] agreed upon, absolute time. Everyone on the planet thinks it's the same time right now in UTC… even though they have different local times.
- **GMT**: Greenwich Mean Time, effectively the same as UTC[2]. You probably want to say UTC, or ``universal time". If you're talking specifically about the GMT time zone, say GMT. Confusingly, many of C's UTC functions predate UTC and still refer to Greenwich Mean Time. When you see that, know that C means UTC.
- **Local time**: what time it is where the computer running the program is located. This is described as an offset from UTC. Although there are many time zones in the world, most computers do work in either local time or UTC.

As a general rule, if you are describing an event that happens one time, like a log entry, or a rocket launch, or when pointers finally clicked for you, use UTC.

On the other hand, if it's something that happens the same time *in every time zone*, like New Year's Eve or dinner time, use local time.

Since a lot of languages are only good at converting between UTC and local time, you can cause yourself a lot of pain by choosing to store your dates in the wrong form. (Ask me how I know.)

[1] On Earth, anyway. Who know what crazy systems they use *out there*…

[2] OK, don't murder me! GMT is technically a time zone while UTC is a global time system. Also some countries might adjust GMT for daylight saving time, whereas UTC is never adjusted for daylight saving time.

38.2 Date Types

There are two[3] main types in C when it comes to dates: `time_t` and `struct tm`.

The spec doesn't actually say much about them:

- `time_t`: a real type capable of holding a time. So by the spec, this could be a floating type or integer type. In POSIX (Unix-likes), it's an integer. This holds *calendar time*. Which you can think of as UTC time.
- `struct tm`: holds the components of a calendar time. This is a *broken-down time*, i.e. the components of the time, like hour, minute, second, day, month, year, etc.

On a lot of systems, `time_t` represents the number of seconds since *Epoch*[4]. Epoch is in some ways the start of time from the computer's perspective, which is commonly January 1, 1970 UTC. `time_t` can go negative to represent times before Epoch. Windows behaves the same way as Unix from what I can tell.

And what's in a `struct tm`? The following fields:

```
struct tm {
    int tm_sec;    // seconds after the minute -- [0, 60]
    int tm_min;    // minutes after the hour -- [0, 59]
    int tm_hour;   // hours since midnight -- [0, 23]
    int tm_mday;   // day of the month -- [1, 31]
    int tm_mon;    // months since January -- [0, 11]
    int tm_year;   // years since 1900
    int tm_wday;   // days since Sunday -- [0, 6]
    int tm_yday;   // days since January 1 -- [0, 365]
    int tm_isdst;  // Daylight Saving Time flag
};
```

Note that everything is zero-based except the day of the month.

It's important to know that you can put any values in these types you want. There are functions to help get the time *now*, but the types hold *a* time, not *the* time.

So the question becomes: ``How do you initialize data of these types, and how do you convert between them?"

38.3 Initialization and Conversion Between Types

First, you can get the current time and store it in a `time_t` with the `time()` function.

```
time_t now;  // Variable to hold the time now

now = time(NULL);  // You can get it like this...

time(&now);        // ...or this. Same as the previous line.
```

Great! You have a variable that gets you the time now.

Amusingly, there's only one portable way to print out what's in a `time_t`, and that's the rarely-used `ctime()` function that prints the value in local time:

```
now = time(NULL);
printf("%s", ctime(&now));
```

[3] Admittedly, there are more than two.

[4] https://en.wikipedia.org/wiki/Unix_time

This returns a string with a very specific form that includes a newline at the end:

```
Sun Feb 28 18:47:25 2021
```

So that's kind of inflexible. If you want more control, you should convert that `time_t` into a `struct tm`.

38.3.1 Converting `time_t` to `struct tm`

There are two amazing ways to do this conversion:

- `localtime()`: this function converts a `time_t` to a `struct tm` in local time.
- `gmtime()`: this function converts a `time_t` to a `struct tm` in UTC. (See ye olde GMT creeping into that function name?)

Let's see what time it is now by printing out a `struct tm` with the `asctime()` function:

```
printf("Local: %s", asctime(localtime(&now)));
printf("  UTC: %s", asctime(gmtime(&now)));
```

Output (I'm in the Pacific Standard Time zone):

```
Local: Sun Feb 28 20:15:27 2021
  UTC: Mon Mar  1 04:15:27 2021
```

Once you have your `time_t` in a `struct tm`, it opens all kinds of doors. You can print out the time in a variety of ways, figure out which day of the week a date is, and so on. Or convert it back into a `time_t`.

More on that soon!

38.3.2 Converting `struct tm` to `time_t`

If you want to go the other way, you can use `mktime()` to get that information.

`mktime()` sets the values of `tm_wday` and `tm_yday` for you, so don't bother filling them out because they'll just be overwritten.

Also, you can set `tm_isdst` to `-1` to have it make the determination for you. Or you can manually set it to true or false.

```
// Don't be tempted to put leading zeros on these numbers (unless you
// mean for them to be in octal)!

struct tm some_time = {
    .tm_year=82,   // years since 1900
    .tm_mon=3,     // months since January -- [0, 11]
    .tm_mday=12,   // day of the month -- [1, 31]
    .tm_hour=12,   // hours since midnight -- [0, 23]
    .tm_min=0,     // minutes after the hour -- [0, 59]
    .tm_sec=4,     // seconds after the minute -- [0, 60]
    .tm_isdst=-1,  // Daylight Saving Time flag
};

time_t some_time_epoch;

some_time_epoch = mktime(&some_time);

printf("%s", ctime(&some_time_epoch));
printf("Is DST: %d\n", some_time.tm_isdst);
```

Output:

```
Mon Apr 12 12:00:04 1982
Is DST: 0
```

When you manually load a `struct tm` like that, it should be in local time. `mktime()` will convert that local time into a `time_t` calendar time.

Weirdly, however, the standard doesn't give us a way to load up a `struct tm` with a UTC time and convert that to a `time_t`. If you want to do that with Unix-likes, try the non-standard `timegm()`. On Windows, `_mkgmtime()`.

38.4 Formatted Date Output

We've already seen a couple ways to print formatted date output to the screen. With `time_t` we can use `ctime()`, and with `struct tm` we can use `asctime()`.

```
time_t now = time(NULL);
struct tm *local = localtime(&now);
struct tm *utc = gmtime(&now);

printf("Local time: %s", ctime(&now));     // Local time with time_t
printf("Local time: %s", asctime(local));  // Local time with struct tm
printf("UTC       : %s", asctime(utc));    // UTC with a struct tm
```

But what if I told you, dear reader, that there's a way to have much more control over how the date was printed?

Sure, we could fish individual fields out of the `struct tm`, but there's a great function called `strftime()` that will do a lot of the hard work for you. It's like `printf()`, except for dates!

Let's see some examples. In each of these, we pass in a destination buffer, a maximum number of characters to write, and then a format string (in the style of---but not the same as---`printf()`) which tells `strftime()` which components of a `struct tm` to print and how.

You can add other constant characters to include in the output in the format string, as well, just like with `printf()`.

We get a `struct tm` in this case from `localtime()`, but any source works fine.

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

int main(void)
{
    char s[128];
    time_t now = time(NULL);

    // %c: print date as per current locale
    strftime(s, sizeof s, "%c", localtime(&now));
    puts(s);   // Sun Feb 28 22:29:00 2021

    // %A: full weekday name
    // %B: full month name
    // %d: day of the month
    strftime(s, sizeof s, "%A, %B %d", localtime(&now));
    puts(s);   // Sunday, February 28

    // %I: hour (12 hour clock)
```

```
    // %M: minute
    // %S: second
    // %p: AM or PM
    strftime(s, sizeof s, "It's %I:%M:%S %p", localtime(&now));
    puts(s);   // It's 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 time zone offset
    strftime(s, sizeof s, "ISO 8601: %FT%T%z", localtime(&now));
    puts(s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

There are a *ton* of date printing format specifiers for `strftime()`, so be sure to check them out in the `strftime()` reference page[5].

38.5 More Resolution with `timespec_get()`

You can get the number of seconds and nanoseconds since Epoch with `timespec_get()`.

Maybe.

Implementations might not have nanosecond resolution (that's one billionth of a second) so who knows how many significant places you'll get, but give it a shot and see.

`timespec_get()` takes two arguments. One is a pointer to a `struct timespec` to hold the time information. And the other is the `base`, which the spec lets you set to `TIME_UTC` indicating that you're interested in seconds since Epoch. (Other implementations might give you more options for the `base`.)

And the structure itself has two fields:

```
struct timespec {
    time_t tv_sec;   // Seconds
    long   tv_nsec;  // Nanoseconds (billionths of a second)
};
```

Here's an example where we get the time and print it out both as integer values and also a floating value:

```
struct timespec ts;

timespec_get(&ts, TIME_UTC);

printf("%ld s, %ld ns\n", ts.tv_sec, ts.tv_nsec);

double float_time = ts.tv_sec + ts.tv_nsec/1000000000.0;
printf("%f seconds since epoch\n", float_time);
```

Example output:

```
1614581530 s, 806325800 ns
1614581530.806326 seconds since epoch
```

`struct timespec` also makes an appearance in a number of the threading functions that need to be able to specify time with that resolution.

[5] https://beej.us/guide/bgclr/html/split/time.html#man-strftime

38.6 Differences Between Times

One quick note about getting the difference between two `time_t`s: since the spec doesn't dictate how that type represents a time, you might not be able to simply subtract two `time_t`s and get anything sensible[6].

Luckily you can use `difftime()` to compute the difference in seconds between two dates.

In the following example, we have two events that occur some time apart, and we use `difftime()` to compute the difference.

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

int main(void)
{
    struct tm time_a = {
        .tm_year=82,   // years since 1900
        .tm_mon=3,     // months since January -- [0, 11]
        .tm_mday=12,   // day of the month -- [1, 31]
        .tm_hour=4,    // hours since midnight -- [0, 23]
        .tm_min=00,    // minutes after the hour -- [0, 59]
        .tm_sec=04,    // seconds after the minute -- [0, 60]
        .tm_isdst=-1,  // Daylight Saving Time flag
    };

    struct tm time_b = {
        .tm_year=120,  // years since 1900
        .tm_mon=10,    // months since January -- [0, 11]
        .tm_mday=15,   // day of the month -- [1, 31]
        .tm_hour=16,   // hours since midnight -- [0, 23]
        .tm_min=27,    // minutes after the hour -- [0, 59]
        .tm_sec=00,    // seconds after the minute -- [0, 60]
        .tm_isdst=-1,  // Daylight Saving Time flag
    };

    time_t cal_a = mktime(&time_a);
    time_t cal_b = mktime(&time_b);

    double diff = difftime(cal_b, cal_a);

    double years = diff / 60 / 60 / 24 / 365.2425;  // close enough

    printf("%f seconds (%f years) between events\n", diff, years);
}
```

Output:

```
1217996816.000000 seconds (38.596783 years) between events
```

And there you have it! Remember to use `difftime()` to take the time difference. Even though you can just subtract on a POSIX system, might as well stay portable.

[6] You will on POSIX, where `time_t` is definitely an integer. Unfortunately the entire world isn't POSIX, so there we are.

Chapter 39

Multithreading

C11 introduced, formally, multithreading to the C language. It's very eerily similar to POSIX threads[1], if you've ever used those.

And if you've not, no worries. We'll talk it through.

Do note, however, that I'm not intending this to be a full-blown classic multithreading how-to[2]; you'll have to pick up a different very thick book for that, specifically. Sorry!

Threading is an optional feature. If a C11+ compiler defines `__STDC_NO_THREADS__`, threads will **not** be present in the library. Why they decided to go with a negative sense in that macro is beyond me, but there we are.

You can test for it like this:

```
#ifdef __STDC_NO_THREADS__
#error I need threads to build this program!
#endif
```

Also, you might need to specify certain linker options when building. In the case of Unix-likes, try appending a `-lpthreads` to the end of the command line to link the `pthreads` library[3]:

```
gcc -std=c11 -o foo foo.c -lpthreads
```

If you're getting linker errors on your system, it could be because the appropriate library wasn't included.

39.1 Background

Threads are a way to have all those shiny CPU cores you paid for do work for you in the same program.

Normally, a C program just runs on a single CPU core. But if you know how to split up the work, you can give pieces of it to a number of threads and have them do the work simultaneously.

Though the spec doesn't say it, on your system it's very likely that C (or the OS at its behest) will attempt to balance the threads over all your CPU cores.

And if you have more threads than cores, that's OK. You just won't realize all those gains if they're all trying to compete for CPU time.

[1] https://en.wikipedia.org/wiki/POSIX_Threads

[2] I'm more a fan of shared-nothing, myself, and my skills with classic multithreading constructs are rusty, to say the least.

[3] Yes, `pthreads` with a ``p". It's short for POSIX threads, a library that C11 borrowed liberally from for its threads implementation.

39.2 Things You Can Do

You can create a thread. It will begin running the function you specify. The parent thread that spawned it will also continue to run.

And you can wait for the thread to complete. This is called *joining*.

Or if you don't care when the thread completes and don't want to wait, you can *detach it*.

A thread can explicitly *exit*, or it can implicitly call it quits by returning from its main function.

A thread can also *sleep* for a period of time, doing nothing while other threads run.

The `main()` program is a thread, as well.

Additionally, we have thread local storage, mutexes, and conditional variables. But more on those later. Let's just look at the basics for now.

39.3 Data Races and the Standard Library

Some of the functions in the standard library (e.g. `asctime()` and `strtok()`) return or use `static` data elements that aren't threadsafe. But in general unless it's said otherwise, the standard library makes an effort to be so[4].

But keep an eye out. If a standard library function is maintaining state between calls in a variable you don't own, or if a function is returning a pointer to a thing that you didn't pass in, it's not threadsafe.

39.4 Creating and Waiting for Threads

Let's hack something up!

We'll make some threads (create) and wait for them to complete (join).

We have a tiny bit to understand first, though.

Every single thread is identified by an opaque variable of type `thrd_t`. It's a unique identifier per thread in your program. When you create a thread, it's given a new ID.

Also when you make the thread, you have to give it a pointer to a function to run, and a pointer to an argument to pass to it (or `NULL` if you don't have anything to pass).

The thread will begin execution on the function you specify.

When you want to wait for a thread to complete, you have to specify its thread ID so C knows which one to wait for.

So the basic idea is:

1. Write a function to act as the thread's ``main''. It's not `main()`-proper, but analogous to it. The thread will start running there.
2. From the main thread, launch a new thread with `thrd_create()`, and pass it a pointer to the function to run.
3. In that function, have the thread do whatever it has to do.
4. Meantimes, the main thread can continue doing whatever *it* has to do.
5. When the main thread decides to, it can wait for the child thread to complete by calling `thrd_join()`. Generally you **must** `thrd_join()` the thread to clean up after it or else you'll leak memory[5]

[4] Per §7.1.4¶5.

[5] Unless you `thrd_detach()`. More on this later.

`thrd_create()` takes a pointer to the function to run, and it's of type `thrd_start_t`, which is `int (*)(void *)`. That's Greek for ``a pointer to a function that takes an `void*` as an argument, and returns an `int`."

Let's make a thread! We'll launch it from the main thread with `thrd_create()` to run a function, do some other things, then wait for it to complete with `thrd_join()`. I've named the thread's main function `run()`, but you can name it anything as long as the types match `thrd_start_t`.

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

// This is the function the thread will run. It can be called anything.
//
// arg is the argument pointer passed to `thrd_create()`.
//
// The parent thread will get the return value back from `thrd_join()`'
// later.

int run(void *arg)
{
    int *a = arg;  // We'll pass in an int* from thrd_create()

    printf("THREAD: Running thread with arg %d\n", *a);

    return 12;  // Value to be picked up by thrd_join() (chose 12 at random)
}

int main(void)
{
    thrd_t t;  // t will hold the thread ID
    int arg = 3490;

    printf("Launching a thread\n");

    // Launch a thread to the run() function, passing a pointer to 3490
    // as an argument. Also stored the thread ID in t:

    thrd_create(&t, run, &arg);

    printf("Doing other things while the thread runs\n");

    printf("Waiting for thread to complete...\n");

    int res;  // Holds return value from the thread exit

    // Wait here for the thread to complete; store the return value
    // in res:

    thrd_join(t, &res);

    printf("Thread exited with return value %d\n", res);
}
```

See how we did the `thrd_create()` there to call the `run()` function? Then we did other things in `main()` and then stopped and waited for the thread to complete with `thrd_join()`.

Sample output (yours might vary):

```
Launching a thread
Doing other things while the thread runs
Waiting for thread to complete...
THREAD: Running thread with arg 3490
Thread exited with return value 12
```

The `arg` that you pass to the function has to have a lifetime long enough so that the thread can pick it up before it goes away. Also, it needs to not be overwritten by the main thread before the new thread can use it.

Let's look at an example that launches 5 threads. One thing to note here is how we use an array of `thrd_t`s to keep track of all the thread IDs.

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

int run(void *arg)
{
    int i = *(int*)arg;

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++)

        // NOTE! In the following line, we pass a pointer to i,
        // but each thread sees the same pointer. So they'll
        // print out weird things as i changes value here in
        // the main thread! (More in the text, below.)

        thrd_create(t + i, run, &i);

    printf("Doing other things while the thread runs...\n");
    printf("Waiting for thread to complete...\n");

    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d complete!\n", res);
    }

    printf("All threads complete!\n");
}
```

When I run the threads, I count `i` up from 0 to 4. And pass a pointer to it to `thrd_create()`. This pointer ends up in the `run()` routine where we make a copy of it.

Simple enough? Here's the output:

```
Launching threads...
THREAD 2: running!
THREAD 3: running!
THREAD 4: running!
THREAD 2: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 2 complete!
Thread 2 complete!
THREAD 5: running!
Thread 3 complete!
Thread 4 complete!
Thread 5 complete!
All threads complete!
```

Whaaa---? Where's `THREAD 0`? And why do we have a `THREAD 5` when clearly `i` is never more than `4` when we call `thrd_create()`? And two `THREAD 2`s? Madness!

This is getting into the fun land of *race conditions*. The main thread is modifying `i` before the thread has a chance to copy it. Indeed, `i` makes it all the way to `5` and ends the loop before the last thread gets a chance to copy it.

We've got to have a per-thread variable that we can refer to so we can pass it in as the `arg`.

We could have a big array of them. Or we could `malloc()` space (and free it somewhere---maybe in the thread itself.)

Let's give that a shot:

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

int run(void *arg)
{
    int i = *(int*)arg;  // Copy the arg

    free(arg);  // Done with this

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
```

```
25      for (i = 0; i < THREAD_COUNT; i++) {
26
27          // Get some space for a per-thread argument:
28
29          int *arg = malloc(sizeof *arg);
30          *arg = i;
31
32          thrd_create(t + i, run, arg);
33      }
34
35      // ...
```

Notice on lines 27-30 we `malloc()` space for an `int` and copy the value of `i` into it. Each new thread gets its own freshly-`malloc()`d variable and we pass a pointer to that to the `run()` function.

Once `run()` makes its own copy of the `arg` on line 7, it `free()`s the `malloc()`d `int`. And now that it has its own copy, it can do with it what it pleases.

And a run shows the result:

```
Launching threads...
THREAD 0: running!
THREAD 1: running!
THREAD 2: running!
THREAD 3: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 0 complete!
Thread 1 complete!
Thread 2 complete!
Thread 3 complete!
THREAD 4: running!
Thread 4 complete!
All threads complete!
```

There we go! Threads 0-4 all in effect!

Your run might vary---how the threads get scheduled to run is beyond the C spec. We see in the above example that thread 4 didn't even begin until threads 0-1 had completed. Indeed, if I run this again, I likely get different output. We cannot guarantee a thread execution order.

39.5 Detaching Threads

If you want to fire-and-forget a thread (i.e. so you don't have to `thrd_join()` it later), you can do that with `thrd_detach()`.

This removes the parent thread's ability to get the return value from the child thread, but if you don't care about that and just want threads to clean up nicely on their own, this is the way to go.

Basically we're going to do this:

```
thrd_create(&t, run, NULL);
thrd_detach(t);
```

where the `thrd_detach()` call is the parent thread saying, ``Hey, I'm not going to wait for this child thread to complete with `thrd_join()`. So go ahead and clean it up on your own when it completes."

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

int run(void *arg)
{
    (void)arg;

    //printf("Thread running! %lu\n", thrd_current()); // non-portable!
    printf("Thread running!\n");

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t;

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(&t, run, NULL);
        thrd_detach(t);              // <-- DETACH!
    }

    // Sleep for a second to let all the threads finish
    thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
}
```

Note that in this code, we put the main thread to sleep for 1 second with `thrd_sleep()`---more on that later.

Also in the `run()` function, I have a commented-out line in there that prints out the thread ID as an `unsigned long`. This is non-portable, because the spec doesn't say what type a `thrd_t` is under the hood---it could be a `struct` for all we know. But that line works on my system.

Something interesting I saw when I ran the code, above, and printed out the thread IDs was that some threads had duplicate IDs! This seems like it should be impossible, but C is allowed to *reuse* thread IDs after the corresponding thread has exited. So what I was seeing was that some threads completed their run before other threads were launched.

39.6 Thread Local Data

Threads are interesting because they don't have their own memory beyond local variables. If you want a `static` variable or file scope variable, all threads will see that same variable.

This can lead to race conditions, where you get *Weird Things*™ happening.

Check out this example. We have a `static` variable `foo` in block scope in `run()`. This variable will be visible to all threads that pass through the `run()` function. And the various threads can effectively step on each others toes.

Each thread copies `foo` into a local variable `x` (which is not shared between threads---all the threads have their own call stacks). So they *should* be the same, right?

And the first time we print them, they are[6]. But then right after that, we check to make sure they're still the

[6]Though I don't think they have to be. It's just that the threads don't seem to get rescheduled until some system call like might happen with a `printf()`… which is why I have the `printf()` in there.

same.

And they *usually* are. But not always!

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

int run(void *arg)
{
    int n = *(int*)arg;  // Thread number for humans to differentiate

    free(arg);

    static int foo = 10;  // Static value shared between threads

    int x = foo;  // Automatic local variable--each thread has its own

    // We just assigned x from foo, so they'd better be equal here.
    // (In all my test runs, they were, but even this isn't guaranteed!)

    printf("Thread %d: x = %d, foo = %d\n", n, x, foo);

    // And they should be equal here, but they're not always!
    // (Sometimes they were, sometimes they weren't!)

    // What happens is another thread gets in and increments foo
    // right now, but this thread's x remains what it was before!

    if (x != foo) {
        printf("Thread %d: Craziness! x != foo! %d != %d\n", n, x, foo);
    }

    foo++;  // Increment shared value

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

Here's an example output (though this varies from run to run):

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 1: Craziness! x != foo! 10 != 11
Thread 2: x = 12, foo = 12
Thread 4: x = 13, foo = 13
Thread 3: x = 14, foo = 14
```

In thread 1, between the two `printf()`s, the value of `foo` somehow changed from `10` to `11`, even though clearly there's no increment between the `printf()`s!

It was another thread that got in there (probably thread 0, from the look of it) and incremented the value of `foo` behind thread 1's back!

Let's solve this problem two different ways. (If you want all the threads to share the variable *and* not step on each other's toes, you'll have to read on to the mutex section.)

39.6.1 `_Thread_local` Storage-Class

First things first, let's just look at the easy way around this: the `_Thread_local` storage-class.

Basically we're just going to slap this on the front of our block scope `static` variable and things will work! It tells C that every thread should have its own version of this variable, so none of them step on each other's toes.

The `<threads.h>` header defines `thread_local` as an alias to `_Thread_local` so your code doesn't have to look so ugly.

Let's take the previous example and make `foo` into a `thread_local` variable so that we don't share that data.

```
5  int run(void *arg)
6  {
7      int n = *(int*)arg;  // Thread number for humans to differentiate
8
9      free(arg);
10
11     thread_local static int foo = 10;  // <-- No longer shared!!
```

And running we get:

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 2: x = 10, foo = 10
Thread 4: x = 10, foo = 10
Thread 3: x = 10, foo = 10
```

No more weird problems!

One thing: if a `thread_local` variable is block scope, it **must** be `static`. Them's the rules. (But this is OK because non-`static` variables are per-thread already since each thread has it's own non-`static` variables.)

A bit of a lie there: block scope `thread_local` variables can also be `extern`.

39.6.2 Another Option: Thread-Specific Storage

Thread-specific storage (TSS) is another way of getting per-thread data.

One additional feature is that these functions allow you to specify a destructor that will be called on the data when the TSS variable is deleted. Commonly this destructor is `free()` to automatically clean up `malloc()`d per-thread data. Or `NULL` if you don't need to destroy anything.

The destructor is type `tss_dtor_t` which is a pointer to a function that returns `void` and takes a `void*` as an argument (the `void*` points to the data stored in the variable). In other words, it's a `void (*)(void*)`, if that clears it up. Which I admit it probably doesn't. Check out the example, below.

Generally, `thread_local` is probably your go-to, but if you like the destructor idea, then you can make use of that.

The usage is a bit weird in that we need a variable of type `tss_t` to be alive to represent the value on a per thread basis. Then we initialize it with `tss_create()`. Eventually we get rid of it with `tss_delete()`. Note that calling `tss_delete()` doesn't run all the destructors---it's `thrd_exit()` (or returning from the run function) that does that. `tss_delete()` just releases any memory allocated by `tss_create()`.

In the middle, threads can call `tss_set()` and `tss_get()` to set and get the value.

In the following code, we set up the TSS variable before creating the threads, then clean up after the threads.

In the `run()` function, the threads `malloc()` some space for a string and store that pointer in the TSS variable.

When the thread exits, the destructor function (`free()` in this case) is called for *all* the threads.

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

tss_t str;

void some_function(void)
{
    // Retrieve the per-thread value of this string
    char *tss_string = tss_get(str);

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Get this thread's serial number
    free(arg);

    // malloc() space to hold the data for this thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Happy little string

    // Set this TSS variable to point at the string
    tss_set(str, s);

    // Call a function that will get the variable
    some_function();

    return 0;   // Equivalent to thrd_exit(0)
}

#define THREAD_COUNT 15

int main(void)
{
```

```
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // All threads are done, so we're done with this
    tss_delete(str);
}
```

Again, this is kind of a painful way of doing things compared to `thread_local`, so unless you really need that destructor functionality, I'd use that instead.

39.7 Mutexes

If you want to only allow a single thread into a critical section of code at a time, you can protect that section with a mutex[7].

For example, if we had a `static` variable and we wanted to be able to get and set it in two operations without another thread jumping in the middle and corrupting it, we could use a mutex for that.

You can acquire a mutex or release it. If you attempt to acquire the mutex and succeed, you may continue execution. If you attempt and fail (because someone else holds it), you will *block*[8] until the mutex is released.

If multiple threads are blocked waiting for a mutex to be released, one of them will be chosen to run (at random, from our perspective), and the others will continue to sleep.

The gameplan is that first we'll initialize a mutex variable to make it ready to use with `mtx_init()`.

Then subsequent threads can call `mtx_lock()` and `mtx_unlock()` to get and release the mutex.

When we're completely done with the mutex, we can destroy it with `mtx_destroy()`, the logical opposite of `mtx_init()`.

First, let's look at some code that does *not* use a mutex, and endeavors to print out a shared (`static`) serial number and then increment it. Because we're not using a mutex over the getting of the value (to print it) and the setting (to increment it), threads might get in each other's way in that critical section.

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

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // Shared static variable!
```

[7] Short for ``mutual exclusion", AKA a ``lock" on a section of code that only one thread is permitted to execute.

[8] That is, your process will go to sleep.

```
    printf("Thread running! %d\n", serial);

    serial++;

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

When I run this, I get something that looks like this:

```
Thread running! 0
Thread running! 0
Thread running! 0
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Clearly multiple threads are getting in there and running the `printf()` before anyone gets a change to update the `serial` variable.

What we want to do is wrap the getting of the variable and setting of it into a single mutex-protected stretch of code.

We'll add a new variable to represent the mutex of type `mtx_t` in file scope, initialize it, and then the threads can lock and unlock it in the `run()` function.

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

mtx_t serial_mtx;     // <-- MUTEX VARIABLE

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // Shared static variable!

```

```
12      // Acquire the mutex--all threads will block on this call until
13      // they get the lock:
14
15      mtx_lock(&serial_mtx);           // <-- ACQUIRE MUTEX
16
17      printf("Thread running! %d\n", serial);
18
19      serial++;
20
21      // Done getting and setting the data, so free the lock. This will
22      // unblock threads on the mtx_lock() call:
23
24      mtx_unlock(&serial_mtx);         // <-- RELEASE MUTEX
25
26      return 0;
27  }
28
29  #define THREAD_COUNT 10
30
31  int main(void)
32  {
33      thrd_t t[THREAD_COUNT];
34
35      // Initialize the mutex variable, indicating this is a normal
36      // no-frills, mutex:
37
38      mtx_init(&serial_mtx, mtx_plain);        // <-- CREATE MUTEX
39
40      for (int i = 0; i < THREAD_COUNT; i++) {
41          thrd_create(t + i, run, NULL);
42      }
43
44      for (int i = 0; i < THREAD_COUNT; i++) {
45          thrd_join(t[i], NULL);
46      }
47
48      // Done with the mutex, destroy it:
49
50      mtx_destroy(&serial_mtx);                // <-- DESTROY MUTEX
51  }
```

See how on lines 38 and 50 of `main()` we initialize and destroy the mutex.

But each individual thread acquires the mutex on line 15 and releases it on line 24.

In between the `mtx_lock()` and `mtx_unlock()` is the *critical section*, the area of code where we don't want multiple threads mucking about at the same time.

And now we get proper output!

```
Thread running! 0
Thread running! 1
Thread running! 2
Thread running! 3
Thread running! 4
Thread running! 5
```

```
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

If you need multiple mutexes, no problem: just have multiple mutex variables.

And always remember the Number One Rule of Multiple Mutexes: *Unlock mutexes in the opposite order in which you lock them!*

39.7.1 Different Mutex Types

As hinted earlier, we have a few mutex types that you can create with `mtx_init()`. (Some of these types are the result of a bitwise-OR operation, as noted in the table.)

Type	Description
`mtx_plain`	Regular ol' mutex
`mtx_timed`	Mutex that supports timeouts
`mtx_plain|mtx_recursive`	Recursive mutex
`mtx_timed|mtx_recursive`	Recursive mutex that supports timeouts

``Recursive'' means that the holder of a lock can call `mtx_lock()` multiple times on the same lock. (They have to unlock it an equal number of times before anyone else can take the mutex.) This might ease coding from time to time, especially if you call a function that needs to lock the mutex when you already hold the mutex.

And the timeout gives a thread a chance to *try* to get the lock for a while, but then bail out if it can't get it in that timeframe.

For a timeout mutex, be sure to create it with `mtx_timed`:

```
mtx_init(&serial_mtx, mtx_timed);
```

And then when you wait for it, you have to specify a time in UTC when it will unlock[9].

The function `timespec_get()` from `<time.h>` can be of assistance here. It'll get you the current time in UTC in a `struct timespec` which is just what we need. In fact, it seems to exist merely for this purpose.

It has two fields: `tv_sec` has the current time in seconds since epoch, and `tv_nsec` has the nanoseconds (billionths of a second) as the ``fractional'' part.

So you can load that up with the current time, and then add to it to get a specific timeout.

Then call `mtx_timedlock()` instead of `mtx_lock()`. If it returns the value `thrd_timedout`, it timed out.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Get current time
timeout.tv_sec += 1;               // Timeout 1 second after now

int result = mtx_timedlock(&serial_mtx, &timeout));

if (result == thrd_timedout) {
    printf("Mutex lock timed out!\n");
}
```

Other than that, timed locks are the same as regular locks.

[9] You might have expected it to be ``time from now'', but you'd just like to think that, wouldn't you!

39.8 Condition Variables

Condition Variables are the last piece of the puzzle we need to make performant multithreaded applications and to compose more complex multithreaded structures.

A condition variable provides a way for threads to go to sleep until some event on another thread occurs.

In other words, we might have a number of threads that are rearing to go, but they have to wait until some event is true before they continue. Basically they're being told ``wait for it!" until they get notified.

And this works hand-in-hand with mutexes since what we're going to wait on generally depends on the value of some data, and that data generally needs to be protected by a mutex.

It's important to note that the condition variable itself isn't the holder of any particular data from our perspective. It's merely the variable by which C keeps track of the waiting/not-waiting status of a particular thread or group of threads.

Let's write a contrived program that reads in groups of 5 numbers from the main thread one at a time. Then, when 5 numbers have been entered, the child thread wakes up, sums up those 5 numbers, and prints the result.

The numbers will be stored in a global, shared array, as will the index into the array of the about-to-be-entered number.

Since these are shared values, we at least have to hide them behind a mutex for both the main and child threads. (The main will be writing data to them and the child will be reading data from them.)

But that's not enough. The child thread needs to block (``sleep") until 5 numbers have been read into the array. And then the parent thread needs to wake up the child thread so it can do its work.

And when it wakes up, it needs to be holding that mutex. And it will! When a thread waits on a condition variable, it also acquires a mutex when it wakes up.

All this takes place around an additional variable of type `cnd_t` that is the *condition variable*. We create this variable with the `cnd_init()` function and destroy it when we're done with it with the `cnd_destroy()` function.

But how's this all work? Let's look at the outline of what the child thread will do:

1. Lock the mutex with `mtx_lock()`
2. If we haven't entered all the numbers, wait on the condition variable with `cnd_wait()`
3. Do the work that needs doing
4. Unlock the mutex with `mtx_unlock()`

Meanwhile the main thread will be doing this:

1. Lock the mutex with `mtx_lock()`
2. Store the recently-read number into the array
3. If the array is full, signal the child to wake up with `cnd_signal()`
4. Unlock the mutex with `mtx_unlock()`

If you didn't skim that too hard (it's OK---I'm not offended), you might notice something weird: how can the main thread hold the mutex lock and signal the child, if the child has to hold the mutex lock to wait for the signal? They can't both hold the lock!

And indeed they don't! There's some behind-the-scenes magic with condition variables: when you `cnd_wait()`, it releases the mutex that you specify and the thread goes to sleep. And when someone signals that thread to wake up, it reacquires the lock as if nothing had happened.

It's a little different on the `cnd_signal()` side of things. This doesn't do anything with the mutex. The signaling thread still must manually release the mutex before the waiting threads can wake up.

One more thing on the `cnd_wait()`. You'll probably be calling `cnd_wait()` if some condition[10] is not yet

[10] And that's why they're called *condition variables*!

met (e.g. in this case, if not all the numbers have yet been entered). Here's the deal: this condition should be in a `while` loop, not an `if` statement. Why?

It's because of a mysterious phenomenon called a *spurious wakeup*. Sometimes, in some implementations, a thread can be woken up out of a `cnd_wait()` sleep for seemingly *no reason*. *[X-Files music]*[11]. And so we have to check to see that the condition we need is still actually met when we wake up. And if it's not, back to sleep with us!

So let's do this thing! Starting with the main thread:

- The main thread will set up the mutex and condition variable, and will launch the child thread.
- Then it will, in an infinite loop, get numbers as input from the console.
- It will also acquire the mutex to store the inputted number into a global array.
- When the array has 5 numbers in it, the main thread will signal the child thread that it's time to wake up and do its work.
- Then the main thread will unlock the mutex and go back to reading the next number from the console.

Meanwhile, the child thread has been up to its own shenanigans:

- The child thread grabs the mutex
- While the condition is not met (i.e. while the shared array doesn't yet have 5 numbers in it), the child thread sleeps by waiting on the condition variable. When it waits, it implicitly unlocks the mutex.
- Once the main thread signals the child thread to wake up, it wakes up to do the work and gets the mutex lock back.
- The child thread sums the numbers and resets the variable that is the index into the array.
- It then releases the mutex and runs again in an infinite loop.

And here's the code! Give it some study so you can see where all the above pieces are being handled:

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

#define VALUE_COUNT_MAX 5

int value[VALUE_COUNT_MAX];  // Shared global
int value_count = 0;   // Shared global, too

mtx_t value_mtx;   // Mutex around value
cnd_t value_cnd;   // Condition variable on value

int run(void *arg)
{
    (void)arg;

    for (;;) {
        mtx_lock(&value_mtx);      // <-- GRAB THE MUTEX

        while (value_count < VALUE_COUNT_MAX) {
            printf("Thread: is waiting\n");
            cnd_wait(&value_cnd, &value_mtx);  // <-- CONDITION WAIT
        }
```

[11] I'm not saying it's aliens… but it's aliens. OK, really more likely another thread might have been woken up and gotten to the work first.

```
        printf("Thread: is awake!\n");

        int t = 0;

        // Add everything up
        for (int i = 0; i < VALUE_COUNT_MAX; i++)
            t += value[i];

        printf("Thread: total is %d\n", t);

        // Reset input index for main thread
        value_count = 0;

        mtx_unlock(&value_mtx);   // <-- MUTEX UNLOCK
    }

    return 0;
}

int main(void)
{
    thrd_t t;

    // Spawn a new thread

    thrd_create(&t, run, NULL);
    thrd_detach(t);

    // Set up the mutex and condition variable

    mtx_init(&value_mtx, mtx_plain);
    cnd_init(&value_cnd);

    for (;;) {
        int n;

        scanf("%d", &n);

        mtx_lock(&value_mtx);    // <-- LOCK MUTEX

        value[value_count++] = n;

        if (value_count == VALUE_COUNT_MAX) {
            printf("Main: signaling thread\n");
            cnd_signal(&value_cnd);  // <-- SIGNAL CONDITION
        }

        mtx_unlock(&value_mtx);  // <-- UNLOCK MUTEX
    }

    // Clean up (I know that's an infinite loop above here, but I
    // want to at least pretend to be proper):

```

```
77      mtx_destroy(&value_mtx);
78      cnd_destroy(&value_cnd);
79  }
```

And here's some sample output (individual numbers on lines are my input):

```
Thread: is waiting
1
1
1
1
1
Main: signaling thread
Thread: is awake!
Thread: total is 5
Thread: is waiting
2
8
5
9
0
Main: signaling thread
Thread: is awake!
Thread: total is 24
Thread: is waiting
```

It's a common use of condition variables in producer-consumer situations like this. If we didn't have a way to put the child thread to sleep while it waited for some condition to be met, it would be force to poll which is a big waste of CPU.

39.8.1 Timed Condition Wait

There's a variant of `cnd_wait()` that allows you to specify a timeout so you can stop waiting.

Since the child thread must relock the mutex, this doesn't necessarily mean that you'll be popping back to life the instant the timeout occurs; you still must wait for any other threads to release the mutex.

But it does mean that you won't be waiting until the `cnd_signal()` happens.

To make this work, call `cnd_timedwait()` instead of `cnd_wait()`. If it returns the value `thrd_timedout`, it timed out.

The timestamp is an absolute time in UTC, not a time-from-now. Thankfully the `timespec_get()` function in `<time.h>` seems custom-made for exactly this case.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Get current time
timeout.tv_sec += 1;               // Timeout 1 second after now

int result = cnd_timedwait(&condition, &mutex, &timeout));

if (result == thrd_timedout) {
    printf("Condition variable timed out!\n");
}
```

39.8.2 Broadcast: Wake Up All Waiting Threads

`cnd_signal()` function]] `cnd_signal()` only wakes up one thread to continue working. Depending on how you have your logic done, it might make sense to wake up more than one thread to continue once the condition is met.

Of course only one of them can grab the mutex, but if you have a situation where:

- The newly-awoken thread is responsible for waking up the next one, and---
- There's a chance the spurious-wakeup loop condition will prevent it from doing so, then---

you'll want to broadcast the wake up so that you're sure to get at least one of the threads out of that loop to launch the next one.

How, you ask?

Simply use `cnd_broadcast()` instead of `cnd_signal()`. Exact same usage, except `cnd_broadcast()` wakes up **all** the sleeping threads that were waiting on that condition variable.

39.9 Running a Function One Time

Let's say you have a function that *could* be run by many threads, but you don't know when, and it's not work trying to write all that logic.

There's a way around it: use `call_once()`. Tons of threads could try to run the function, but only the first one counts[12]

To work with this, you need a special flag variable you declare to keep track of whether or not the thing's been run. And you need a function to run, which takes no parameters and returns no value.

```
once_flag of = ONCE_FLAG_INIT;  // Initialize it like this

void run_once_function(void)
{
    printf("I'll only run once!\n");
}

int run(void *arg)
{
    (void)arg;

    call_once(&of, run_once_function);

    // ...
```

In this example, no matter how many threads get to the `run()` function, the `run_once_function()` will only be called a single time.

[12] Survival of the fittest! Right? I admit it's actually nothing like that.

Chapter 40

Atomics

> ``They tried and failed, all of them?''
> ``Oh, no.'' She shook her head. ``They tried and died.''
>
> ---Paul Atreides and The Reverend Mother Gaius Helen Mohiam, *Dune*

This is one of the more challenging aspects of multithreading with C. But we'll try to take it easy.

Basically, I'll talk about the more straightforward uses of atomic variables, what they are, and how they work, etc. And I'll mention some of the more insanely-complex paths that are available to you.

But I won't go down those paths. Not only am I barely qualified to even write about them, but I figure if you know you need them, you already know more than I do.

But there are some weird things out here even in the basics. So buckle your seatbelts, everyone, `cause Kansas is goin' bye-bye.

40.1 Testing for Atomic Support

Atomics are an optional feature. There's a macro `__STDC_NO_ATOMICS__` that's `1` if you *don't* have atomics.

That macro might not exist pre-C11, so we should test the language version with `__STDC_VERSION__`[1].

```
#if __STDC_VERSION__ < 201112L || __STDC_NO_ATOMICS__ == 1
#define HAS_ATOMICS 0
#else
#define HAS_ATOMICS 1
#endif
```

If those tests pass, then you can safely include `<stdatomic.h>`, the header on which the rest of this chapter is based. But if there is no atomic support, that header might not even exist.

On some systems, you might need to add `-latomic` to the end of your compilation command line to use any functions in the header file.

40.2 Atomic Variables

Here's *part* of how atomic variables work:

If you have a shared atomic variable and you write to it from one thread, that write will be *all-or-nothing* in a different thread.

[1] The `__STDC_VERSION__` macro didn't exist in early C89, so if you're worried about that, check it with `#ifdef`.

That is, the other thread will see the entire write of, say, a 32-bit value. Not half of it. There's no way for one thread to interrupt another that is in the *middle* of an atomic multi-byte write.

It's almost like there's a little lock around the getting and setting of that one variable. (And there *might* be! See Lock-Free Atomic Variables, below.)

And on that note, you can get away with never using atomics if you use mutexes to lock your critical sections. It's just that there are a class of *lock-free data structures* that always allow other threads to make progress instead of being blocked by a mutex… but these are tough to create correctly from scratch, and are one of the things that are beyond the scope of the guide, sadly.

That's only part of the story. But it's the part we'll start with.

Before we go further, how do you declare a variable to be atomic?

First, include `<stdatomic.h>`.

This gives us types such as `atomic_int`.

And then we can simply declare variables to be of that type.

But let's do a demo where we have two threads. The first runs for a while and then sets a variable to a specific value, then exits. The other runs until it sees that value get set, and then it exits.

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

atomic_int x;   // THE POWER OF ATOMICS! BWHAHAHA!

int thread1(void *arg)
{
    (void)arg;

    printf("Thread 1: Sleeping for 1.5 seconds\n");
    thrd_sleep(&(struct timespec){.tv_sec=1, .tv_nsec=500000000}, NULL);

    printf("Thread 1: Setting x to 3490\n");
    x = 3490;

    printf("Thread 1: Exiting\n");
    return 0;
}

int thread2(void *arg)
{
    (void)arg;

    printf("Thread 2: Waiting for 3490\n");
    while (x != 3490) {}  // spin here

    printf("Thread 2: Got 3490--exiting!\n");
    return 0;
}

int main(void)
{
    x = 0;
```

```
35
36      thrd_t t1, t2;
37
38      thrd_create(&t1, thread1, NULL);
39      thrd_create(&t2, thread2, NULL);
40
41      thrd_join(t1, NULL);
42      thrd_join(t2, NULL);
43
44      printf("Main    : Threads are done, so x better be 3490\n");
45      printf("Main    : And indeed, x == %d\n", x);
46  }
```

The second thread spins in place, looking at the flag and waiting for it to get set to the value `3490`. And the first one does that.

And I get this output:

```
Thread 1: Sleeping for 1.5 seconds
Thread 2: Waiting for 3490
Thread 1: Setting x to 3490
Thread 1: Exiting
Thread 2: Got 3490--exiting!
Main    : Threads are done, so x better be 3490
Main    : And indeed, x == 3490
```

Look, ma! We're accessing a variable from different threads and not using a mutex! And that'll work every time thanks to the atomic nature of atomic variables.

You might be wondering what happens if that's a regular non-atomic `int`, instead. Well, on my system it still works… unless I do an optimized build in which case it hangs on thread 2 waiting to see the 3490 to get set[2].

But that's just the beginning of the story. The next part is going to require more brain power and has to do with something called *synchronization*.

40.3 Synchronization

The next part of our story is all about when certain memory writes in one thread become visible to those in another thread.

You might think, it's right away, right? But it's not. A number of things can go wrong. Weirdly wrong.

The compiler might have rearranged memory accesses so that when you think you set a value relative to another might not be true. And even if the compiler didn't, your CPU might have done it on the fly. Or maybe there's something else about this architecture that causes writes on one CPU to be delayed before they're visible on another.

The good news is that we can condense all these potential troubles into one: unsynchronized memory accesses can appear out of order depending on which thread is doing the observing, as if the lines of code themselves had been rearranged.

By way of example, which happens first in the following code, the write to `x` or the write to `y`?

```
1   int x, y;  // global
2
```

[2]The reason for this is when optimized, my compiler has put the value of `x` in a register to make the `while` loop fast. But the register has no way of knowing that the variable was updated in another thread, so it never sees the `3490`. This isn't really related to the *all-or-nothing* part of atomicity, but is more related to the synchronization aspects in the next section.

```
3  // ...
4
5  x = 2;
6  y = 3;
7
8  printf("%d %d\n", x, y);
```

Answer: we don't know. The compiler or CPU could silently reverse lines 5 and 6 and we'd be none-the-wiser. The code would run single-threaded *as-if* it were executed in code order.

In a multithreaded scenario, we might have something like this pseudocode:

```
1  int x = 0, y = 0;
2
3  thread1() {
4      x = 2;
5      y = 3;
6  }
7
8  thread2() {
9      while (y != 3) {}  // spin
10     printf("x is now %d\n", x);  // 2? ...or 0?
11 }
```

What is the output from thread 2?

Well, if `x` gets assigned `2` *before* `y` is assigned `3`, then I'd expect the output to be the very sensible:

```
x is now 2
```

But something sneaky could rearrange lines 4 and 5 causing us to see the value of `0` for `x` when we print it.

In other words, all bets are off unless we can somehow say, ``As of this point, I expect all previous writes in another thread to be visible in this thread."

Two threads *synchronize* when they agree on the state of shared memory. As we've seen, they're not always in agreement with the code. So how do they agree?

Using atomic variables can force the agreement[3]. If a thread writes to an atomic variable, it's saying ``anyone who reads this atomic variable in the future will also see all the changes I made to memory (atomic or not) up to and including the atomic variable".

Or, in more human terms, let's sit around the conference table and make sure we're on the same page as to which pieces of shared memory hold what values. You agree that the memory changes that you'd made up-to-and-including the atomic store will be visible to me after I do a load of the same atomic variable.

So we can easily fix our example:

```
1  int x = 0;
2  atomic int y = 0;  // Make y atomic
3
4  thread1() {
5      x = 2;
6      y = 3;             // Synchronize on write
7  }
8
9  thread2() {
10     while (y != 3) {}  // Synchronize on read
```

[3]Until I say otherwise, I'm speaking generally about *sequentially consistent* operations. More on what that means soon.

```
11      printf("x is now %d\n", x);  // 2, period.
12  }
```

Because the threads synchronize across `y`, all writes in thread 1 that happened *before* the write to `y` are visible in thread 2 *after* the read from `y` (in the `while` loop).

It's important to note a couple things here:

1. Nothing sleeps. The synchronization is not a blocking operation. Both threads are running full bore until they exit. Even the one stuck in the spin loop isn't blocking anyone else from running.
2. The synchronization happens when one thread reads an atomic variable another thread wrote. So when thread 2 reads `y`, all previous memory writes in thread 1 (namely setting `x`) will be visible in thread 2.
3. Notice that `x` isn't atomic. That's OK because we're not synchronizing over `x`, and the synchronization over `y` when we write it in thread 1 means that all previous writes---including `x`---in thread 1 will become visible to other threads… if those other threads read `y` to synchronize.

Forcing this synchronization is inefficient and can be a lot slower than just using a regular variable. This is why we don't use atomics unless we have to for a particular application.

So that's the basics. Let's look deeper.

40.4 Acquire and Release

More terminology! It'll pay off to learn this now.

When a thread reads an atomic variable, it is said to be an *acquire* operation.

When a thread writes an atomic variable, it is said to be a *release* operation.

What are these? Let's line them up with terms you already know when it comes to atomic variables:

Read = Load = Acquire. Like when you compare an atomic variable or read it to copy it to another value.

Write = Store = Release. Like when you assign a value into an atomic variable.

When using atomic variables with these acquire/release semantics, C spells out what can happen when.

Acquire/release form the basis for the synchronization we just talked about.

When a thread acquires an atomic variable, it can see values set in another thread that released that same variable.

In other words:

When a thread reads an atomic variable, it can see values set in another thread that wrote to that same variable.

The synchronization happens across the acquire/release pair.

More details:

With read/load/acquire of a particular atomic variable:

- All writes (atomic or non-atomic) in another thread that happened before that other thread wrote/stored/released this atomic variable are now visible in this thread.
- The new value of the atomic variable set by the other thread is also visible in this thread.
- No reads or writes of any variables/memory in the current thread can be reordered to happen before this acquire.
- The acquire acts as a one-way barrier when it comes to code reordering; reads and writes in the current thread can be moved down from *before* the acquire to *after* it. But, more importantly for synchronization, nothing can move up from *after* the acquire to *before* it.

With write/store/release of a particular atomic variable:

- All writes (atomic or non-atomic) in the current thread that happened before this release become visible to other threads that have read/loaded/acquired the same atomic variable.
- The value written to this atomic variable by this thread is also visible to other threads.
- No reads or writes of any variables/memory in the current thread can be reordered to happen after this release.
- The release acts as a one-way barrier when it comes to code reordering: reads and writes in the current thread can be moved up from *after* the release to *before* it. But, more importantly for synchronization, nothing can move down from *before* the release to *after* it.

Again, the upshot is synchronization of memory from one thread to another. The second thread can be sure that variables and memory are written in the order the programmer intended.

```
int x, y, z = 0;
atomic_int a = 0;

thread1() {
    x = 10;
    y = 20;
    a = 999;  // Release
    z = 30;
}

thread2()
{
    while (a != 999) { } // Acquire

    assert(x == 10);  // never asserts, x is always 10
    assert(y == 20);  // never asserts, y is always 20

    assert(z == 0);  // might assert!!
}
```

In the above example, `thread2` can be sure of the values in `x` and `y` after it acquires `a` because they were set before `thread1` released the atomic `a`.

But `thread2` can't be sure of `z`'s value because it happened after the release. Maybe the assignment to `z` got moved before the assignment to `a`.

An important note: releasing one atomic variable has no effect on acquires of different atomic variables. Each variable is isolated from the others.

40.5 Sequential Consistency

You hanging in there? We're through the meat of the simpler usage of atomics. And since we're not even going to talk about the more complex uses here, you can relax a bit.

Sequential consistency is what's called a *memory ordering*. There are many memory orderings, but sequential consistency is the sanest[4] C has to offer. It is also the default. You have to go out of your way to use other memory orderings.

All the stuff we've been talking about so far has happened within the realm of sequential consistency.

[4]Sanest from a programmer perspective.

We've talked about how the compiler or CPU can rearrange memory reads and writes in a single thread as long as it follows the *as-if* rule.

And we've seen how we can put the brakes on this behavior by synchronizing over atomic variables.

Let's formalize just a little more.

If operations are *sequentially consistent*, it means at the end of the day, when all is said and done, all the threads can kick up their feet, open their beverage of choice, and all agree on the order in which memory changes occurred during the run. And that order is the one specified by the code.

One won't say, ``But didn't *B* happen before *A*?" if the rest of them say, ``*A* definitely happened before *B*". They're all friends, here.

In particular, within a thread, none of the acquires and releases can be reordered with respect to one another. This is in addition to the rules about what other memory accesses can be reordered around them.

This rule gives an additional level of sanity to the progression of atomic loads/acquires and stores/releases.

Every other memory order in C involves a relaxation of the reordering rules, either for acquires/releases or other memory accesses, atomic or otherwise. You'd do that if you *really* knew what you were doing and needed the speed boost. *Here be armies of dragons…*

More on that later, but for now, let's stick to the safe and practical.

40.6 Atomic Assignments and Operators

Certain operators on atomic variables are atomic. And others aren't.

Let's start with a counter-example:

```
atomic_int x = 0;

thread1() {
    x = x + 3;  // NOT atomic!
}
```

Since there's a read of `x` on the right hand side of the assignment and a write effectively on the left, these are two operations. Another thread could sneak in the middle and make you unhappy.

But you *can* use the shorthand `+=` to get an atomic operation:

```
atomic_int x = 0;

thread1() {
    x += 3;   // ATOMIC!
}
```

In that case, `x` will be atomically incremented by `3`---no other thread can jump in the middle.

In particular, the following operators are atomic read-modify-write operations with sequential consistency, so use them with gleeful abandon. (In the example, `a` is atomic.)

```
a++       a--       --a       ++a
a += b    a -= b    a *= b    a /= b    a %= b
a &= b    a |= b    a ^= b    a >>= b   a <<= b
```

40.7 Library Functions that Automatically Synchronize

So far we've talked about how you can synchronize with atomic variables, but it turns out there are a few library functions that do some limited behind-the-scenes synchronization, themselves.

```
call_once()       thrd_create()       thrd_join()
mtx_lock()        mtx_timedlock()     mtx_trylock()
malloc()          calloc()            realloc()
aligned_alloc()
```

`call_once()`---Synchronizes with all subsequent calls to `call_once()` for a particular flag. This way subsequent calls can rest assured that if another thread sets the flag, they will see it.

`thrd_create()`---Synchronizes with the beginning of the new thread. The new thread can be sure it will see all shared memory writes from the parent thread from before the `thrd_create()` call.

`thrd_join()`---When a thread dies, it synchronizes with this function. The thread that has called `thrd_join()` can be assured that it can see all the late thread's shared writes.

`mtx_lock()`---Earlier calls to `mtx_unlock()` on the same mutex synchronize on this call. This is the case that most mirrors the acquire/release process we've already talked about. `mtx_unlock()` performs a release on the mutex variable, assuring any subsequent thread that makes an acquire with `mtx_lock()` can see all the shared memory changes in the critical section.

`mtx_timedlock()` and **`mtx_trylock()`**---Similar to the situation with `mtx_lock()`, if this call succeeds, earlier calls to `mtx_unlock()` synchronize with this one.

Dynamic Memory Functions: if you allocate memory, it synchronizes with the previous deallocation of that same memory. And allocations and deallocations of that particular memory region happen in a single total order that all threads can agree upon. I *think* the idea here is that the deallocation can wipe the region if it chooses, and we want to be sure that a subsequent allocation doesn't see the non-wiped data. Someone let me know if there's more to it.

40.8 Atomic Type Specifier, Qualifier

Let's take it down a notch and see what types we have available, and how we can even make new atomic types.

First things first, let's look at the built-in atomic types and what they are `typedef`'d to. (Spoiler: `_Atomic` is a type qualifier!)

Atomic type	Longhand equivalent
`atomic_bool`	`_Atomic _Bool`
`atomic_char`	`_Atomic char`
`atomic_schar`	`_Atomic signed char`
`atomic_uchar`	`_Atomic unsigned char`
`atomic_short`	`_Atomic short`
`atomic_ushort`	`_Atomic unsigned short`
`atomic_int`	`_Atomic int`
`atomic_uint`	`_Atomic unsigned int`
`atomic_long`	`_Atomic long`
`atomic_ulong`	`_Atomic unsigned long`
`atomic_llong`	`_Atomic long long`
`atomic_ullong`	`_Atomic unsigned long long`
`atomic_char16_t`	`_Atomic char16_t`
`atomic_char32_t`	`_Atomic char32_t`

Atomic type	Longhand equivalent
`atomic_wchar_t`	`_Atomic wchar_t`
`atomic_int_least8_t`	`_Atomic int_least8_t`
`atomic_uint_least8_t`	`_Atomic uint_least8_t`
`atomic_int_least16_t`	`_Atomic int_least16_t`
`atomic_uint_least16_t`	`_Atomic uint_least16_t`
`atomic_int_least32_t`	`_Atomic int_least32_t`
`atomic_uint_least32_t`	`_Atomic uint_least32_t`
`atomic_int_least64_t`	`_Atomic int_least64_t`
`atomic_uint_least64_t`	`_Atomic uint_least64_t`
`atomic_int_fast8_t`	`_Atomic int_fast8_t`
`atomic_uint_fast8_t`	`_Atomic uint_fast8_t`
`atomic_int_fast16_t`	`_Atomic int_fast16_t`
`atomic_uint_fast16_t`	`_Atomic uint_fast16_t`
`atomic_int_fast32_t`	`_Atomic int_fast32_t`
`atomic_uint_fast32_t`	`_Atomic uint_fast32_t`
`atomic_int_fast64_t`	`_Atomic int_fast64_t`
`atomic_uint_fast64_t`	`_Atomic uint_fast64_t`
`atomic_intptr_t`	`_Atomic intptr_t`
`atomic_uintptr_t`	`_Atomic uintptr_t`
`atomic_size_t`	`_Atomic size_t`
`atomic_ptrdiff_t`	`_Atomic ptrdiff_t`
`atomic_intmax_t`	`_Atomic intmax_t`
`atomic_uintmax_t`	`_Atomic uintmax_t`

Use those at will! They're consistent with the atomic aliases found in C++, if that helps.

But what if you want more?

You can do it either with a type qualifier or type specifier.

First, specifier! It's the keyword `_Atomic` with a type in parens after[5]---suitable for use with `typedef`:

```
typedef _Atomic(double) atomic_double;

atomic_double f;
```

Restrictions on the specifier: the type you're making atomic can't be of type array or function, nor can it be atomic or otherwise qualified.

Next, qualifier! It's the keyword `_Atomic` *without* a type in parens.

So these do similar things[6]:

```
_Atomic(int) i;   // type specifier
_Atomic int  j;   // type qualifier
```

The thing is, you can include other type qualifiers with the latter:

```
_Atomic volatile int k;   // qualified atomic variable
```

Restrictions on the qualifier: the type you're making atomic can't be of type array or function.

[5]Apparently C++23 is adding this as a macro.

[6]The spec notes that they might differ in size, representation, and alignment.

40.9 Lock-Free Atomic Variables

Hardware architectures are limited in the amount of data they can atomically read and write. It depends on how it's wired together. And it varies.

If you use an atomic type, you can be assured that accesses to that type will be atomic… but there's a catch: if the hardware can't do it, it's done with a lock, instead.

So the atomic access becomes lock-access-unlock, which is rather slower and has some implications for signal handlers.

Atomic flags, below, is the only atomic type that is guaranteed to be lock-free in all conforming implementations. In typical desktop/laptop computer world, other larger types are likely lock-free.

Luckily, we have a couple ways to determine if a particular type is a lock-free atomic or not.

First of all, some macros---you can use these at compile time with `#if`. They apply to both signed and unsigned types.

Atomic Type	Lock Free Macro
`atomic_bool`	`ATOMIC_BOOL_LOCK_FREE`
`atomic_char`	`ATOMIC_CHAR_LOCK_FREE`
`atomic_char16_t`	`ATOMIC_CHAR16_T_LOCK_FREE`
`atomic_char32_t`	`ATOMIC_CHAR32_T_LOCK_FREE`
`atomic_wchar_t`	`ATOMIC_WCHAR_T_LOCK_FREE`
`atomic_short`	`ATOMIC_SHORT_LOCK_FREE`
`atomic_int`	`ATOMIC_INT_LOCK_FREE`
`atomic_long`	`ATOMIC_LONG_LOCK_FREE`
`atomic_llong`	`ATOMIC_LLONG_LOCK_FREE`
`atomic_intptr_t`	`ATOMIC_POINTER_LOCK_FREE`

These macros can interestingly have *three* different values:

Value	Meaning
`0`	Never lock-free.
`1`	*Sometimes* lock-free.
`2`	Always lock-free.

Wait---how can something be *sometimes* lock-free? This just means the answer isn't known at compile-time, but could later be known at runtime. Maybe the answer varies depending on whether or not you're running this code on Genuine Intel or AMD, or something like that[7].

But you can always test at runtime with the `atomic_is_lock_free()` function. This function returns true or false if the particular type is atomic right now.

So why do we care?

Lock-free is faster, so maybe there's a speed concern that you'd code around another way. Or maybe you need to use an atomic variable in a signal handler.

[7] I just pulled that example out of nowhere. Maybe it doesn't matter on Intel/AMD, but it could matter somewhere, dangit!

40.9.1 Signal Handlers and Lock-Free Atomics

If you read or write a shared variable (static storage duration or `_Thread_Local`) in a signal handler, it's undefined behavior [gasp!]… Unless you do one of the following:

1. Write to a variable of type `volatile sig_atomic_t`.
2. Read or write a lock-free atomic variable.

As far as I can tell, lock-free atomic variables are one of the few ways you get portably get information out of a signal handler.

The spec is a bit vague, in my read, about the memory order when it comes to acquiring or releasing atomic variables in the signal handler. C++ says, and it makes sense, that such accesses are unsequenced with respect to the rest of the program[8]. The signal can be raised, after all, at any time. So I'm assuming C's behavior is similar.

40.10 Atomic Flags

There's only one type the standard guarantees will be a lock-free atomic: `atomic_flag`. This is an opaque type for test-and-set[9] operations.

It can be either *set* or *clear*. You can initialize it to clear with:

```
atomic_flag f = ATOMIC_FLAG_INIT;
```

You can set the flag atomically with `atomic_flag_test_and_set()`, which will set the flag and return its previous status as a `_Bool` (true for set).

You can clear the flag atomically with `atomic_flag_clear()`.

Here's an example where we init the flag to clear, set it twice, then clear it again.

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

atomic_flag f = ATOMIC_FLAG_INIT;

int main(void)
{
    bool r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0

    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 1

    atomic_flag_clear(&f);
    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0
}
```

40.11 Atomic `structs` and `unions`

Using the `_Atomic` qualifier or specifier, you can make atomic `structs` or `unions`! Pretty astounding.

[8] C++ elaborates that if the signal is the result of a call to `raise()`, it is sequenced *after* the `raise()`.

[9] https://en.wikipedia.org/wiki/Test-and-set

If there's not a lot of data in there (i.e. a handful of bytes), the resulting atomic type might be lock-free. Test it with `atomic_is_lock_free()`.

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

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;

    printf("Is lock free: %d\n", atomic_is_lock_free(&p));
}
```

Here's the catch: you can't access fields of an atomic `struct` or `union`... so what's the point? Well, you can atomically *copy* the entire `struct` into a non-atomic variable and then use it. You can atomically copy the other way, too.

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

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;
    struct point t;

    p = (struct point){1, 2};  // Atomic copy

    //printf("%f\n", p.x);  // Error

    t = p;   // Atomic copy

    printf("%f\n", t.x);  // OK!
}
```

You can also declare a `struct` where individual fields are atomic. It is implementation defined if atomic types are allowed on bitfields.

40.12 Atomic Pointers

Just a note here about placement of `_Atomic` when it comes to pointers.

First, pointers to atomics (i.e. the pointer value is not atomic, but the thing it points to is):

```
_Atomic int x;
_Atomic int *p;  // p is a pointer to an atomic int

p = &x;  // OK!
```

Second, atomic pointers to non-atomic values (i.e. the pointer value itself is atomic, but the thing it points to is not):

```
int x;
int * _Atomic p;  // p is an atomic pointer to an int

p = &x;  // OK!
```

Lastly, atomic pointers to atomic values (i.e. the pointer and the thing it points to are both atomic):

```
_Atomic int x;
_Atomic int * _Atomic p;  // p is an atomic pointer to an atomic int

p = &x;  // OK!
```

40.13 Memory Order

We've already talked about sequential consistency, which is the sensible one of the bunch. But there are a number of other ones:

`memory_order`	Description
`memory_order_seq_cst`	Sequential Consistency
`memory_order_acq_rel`	Acquire/Release
`memory_order_release`	Release
`memory_order_acquire`	Acquire
`memory_order_consume`	Consume
`memory_order_relaxed`	Relaxed

You can specify other ones with certain library functions. For example, you can add a value to an atomic variable like this:

```
atomic_int x = 0;

x += 5;  // Sequential consistency, the default
```

Or you can do the same with this library function:

```
atomic_int x = 0;

atomic_fetch_add(&x, 5);  // Sequential consistency, the default
```

Or you can do the same thing with an explicit memory ordering:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_seq_cst);
```

But what if we didn't want sequential consistency? And you wanted acquire/release instead for whatever reason? Just name it:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_acq_rel);
```

We'll do a breakdown of the different memory orders, below. Don't mess with anything other than sequential consistency unless you know what you're doing. It's really easy to make mistakes that will cause rare, hard-to-repro failures.

40.13.1 Sequential Consistency

- Load operations acquire (see below).
- Store operations release (see below).
- Read-modify-write operations acquire then release.

Also, in order to maintain the total order of acquires and releases, no acquires or releases will be reordered with respect to each other. (The acquire/release rules do not forbid reordering a release followed by an acquire. But the sequentially consistent rules do.)

40.13.2 Acquire

This is what happens on a load/read operation on an atomic variable.

- If another thread released this atomic variable, all the writes that thread did are now visible in this thread.
- Memory accesses in this thread that happen after this load can't be reordered before it.

40.13.3 Release

This is what happens on a store/write of an atomic variable.

- If another thread later acquires this atomic variable, all memory writes in this thread before its atomic write become visible to that other thread.
- Memory accesses in this thread that happen before the release can't be reordered after it.

40.13.4 Consume

This is an odd one, similar to a less-strict version of acquire. It affects memory accesses that are *data dependent* on the atomic variable.

Being ``data dependent" vaguely means that the atomic variable is used in a calculation.

That is, if a thread consumes an atomic variable then all the operations in that thread that go on to use that atomic variable will be able to see the memory writes in the releasing thread.

Compare to acquire where memory writes in the releasing thread will be visible to *all* operations in the current thread, not just the data-dependent ones.

Also like acquire, there is a restriction on which operations can be reordered *before* the consume. With acquire, you couldn't reorder anything before it. With consume, you can't reorder anything that depends on the loaded atomic value before it.

40.13.5 Acquire/Release

This only applies to read-modify-write operations. It's an acquire and release bundled into one.

- An acquire happens for the read.
- A release happens for the write.

40.13.6 Relaxed

No rules; it's anarchy! Everyone can reorder everything everywhere! Dogs and cats living together---mass hysteria!

Actually, there is a rule. Atomic reads and writes are still all-or-nothing. But the operations can be reordered whimsically and there is zero synchronization between threads.

There are a few use cases for this memory order, which you can find with a tiny bit of searching, e.g. simple counters.

And you can use a fence to force synchronization after a bunch of relaxed writes.

40.14 Fences

You know how the releases and acquires of atomic variables occur as you read and write them?

Well, it's possible to do a release or acquire *without* an atomic variable, as well.

This is called a *fence*. So if you want all the writes in a thread to be visible elsewhere, you can put up a release fence in one thread and an acquire fence in another, just like with how atomic variables work.

Since a consume operation doesn't really make sense on a fence[10], `memory_order_consume` is treated as an acquire.

You can put up a fence with any specified order:

```
atomic_thread_fence(memory_order_release);
```

There's also a light version of a fence for use with signal handlers, called `atomic_signal_fence()`.

It works just the same way as `atomic_thread_fence()`, except:

- It only deals with visibility of values within the same thread; there is no synchronization with other threads.
- No hardware fence instructions are emitted.

If you want to be sure the side effects of non-atomic operations (and relaxed atomic operations) are visible in the signal handler, you can use this fence.

The idea is that the signal handler is executing in *this* thread, not another, so this is a lighter-weight way of making sure changes outside the signal handler are visible within it (i.e. they haven't been reordered).

40.15 References

If you want to learn more about this stuff, here are some of the things that helped me plow through it:

- Herb Sutter's *`atomic`<> Weapons* talk:
  - Part 1[11]
  - part 2[12]
- Jeff Preshing's materials[13], in particular:
  - An Introduction to Lock-Free Programming[14]
  - Acquire and Release Semantics[15]
  - The *Happens-Before* Relation[16]
  - The *Synchronizes-With* Relation[17]
  - The Purpose of `memory_order_consume` in C++11[18]

[10] Because consume is all about the operations that are dependent on the value of the acquired atomic variable, and there is no atomic variable with a fence.

[11] https://www.youtube.com/watch?v=A8eCGOqgvH4

[12] https://www.youtube.com/watch?v=KeLBd2EJLOU

[13] https://preshing.com/archives/

[14] https://preshing.com/20120612/an-introduction-to-lock-free-programming/

[15] https://preshing.com/20120913/acquire-and-release-semantics/

[16] https://preshing.com/20130702/the-happens-before-relation/

[17] https://preshing.com/20130823/the-synchronizes-with-relation/

[18] https://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

  - You Can Do Any Kind of Atomic Read-Modify-Write Operation[19]

- CPPReference:

  - Memory Order[20]
  - Atomic Types[21]

- Bruce Dawson's Lockless Programming Considerations[22]

- The helpful and knowledgeable folks on r/C_Programming[23]

[19] https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/
[20] https://en.cppreference.com/w/c/atomic/memory_order
[21] https://en.cppreference.com/w/c/language/atomic
[22] https://docs.microsoft.com/en-us/windows/win32/dxtecharts/lockless-programming
[23] https://www.reddit.com/r/C_Programming/

Chapter 41

Function Specifiers, Alignment Specifiers/Operators

These don't see a heck of a lot of use in my experience, but we'll cover them here for the sake of completeness.

41.1 Function Specifiers

When you declare a function, you can give the compiler a couple tips about how the functions could or will be used. This enables or encourages the compiler to make certain optimizations.

41.1.1 `inline` for Speed---Maybe

You can declare a function to be inline like this:

```
static inline int add(int x, int y) {
    return x + y;
}
```

This is meant to encourage the compiler to make this function call as fast as possible. And, historically, one way to do this was *inlining*, which means that the body of the function would be embedded in its entirety where the call was made. This would avoid all the overhead of setting up the function call and tearing it down at the expense of larger code size as the function was copied all over the place instead of being reused.

That would seem to be the end of the story, but it's not. `inline` comes with a whole pile of rules that make for *interesting times*. I'm not sure I even understand them all, and behavior seems to vary from compiler to compiler.

The short answer is define the `inline` function as `static` in the file that you need it. And then use it in that one file. And you never have to worry about the rest of it.

But if you're wondering, here are more fun times.

Let's try leaving the `static` off.

```
#include <stdio.h>

inline int add(int x, int y)
{
    return x + y;
}

```

```
8  int main(void)
9  {
10     printf("%d\n", add(1, 2));
11 }
```

`gcc` gives a linker error on `add()`... unless you compile with optimizations on (probably)!

See, a compiler can choose to inline or not, but if it chooses not to, you're left with no function at all. `gcc` doesn't inline unless you're doing an optimized build.

One way around this is to define a non-`inline` external linkage version of the function elsewhere, and that one will be used when the `inline` one isn't. But you as the programmer can't determine which, portably. If both are available, it's unspecified which one the compiler chooses. With `gcc` the inline function will be used if you're compiling with optimizations, and the non-inline one will be used otherwise. Even if the bodies of these functions are completely different. Zany!

Another way is to declare the function as `extern inline`. This will attempt to inline in this file, but will also create a version with external linkage. And so `gcc` will use one or the other depending on optimizations, but at least they're the same function.

Unless, of course, you have another source file with an `inline` function of the same name; it will use its `inline` function or the one with external linkage depending on optimizations.

But let's say you're doing a build where the compiler *is* inlining the function. In that case, you can just use a plain `inline` in the definition. However, there are now additional restrictions.

You can't refer to any `static` globals or call any `static` functions:

```
static int b = 13;

inline int add(int x, int y)
{
    return x + y + b;  // BAD -- can't refer to b
}
```

And you can't define any non-`const` `static` local variables:

```
inline int add(int x, int y)
{
    static int b = 13;  // BAD -- can't define static

    return x + y + b;
}
```

But making it `const` is OK:

```
inline int add(int x, int y)
{
    static const int b = 13;  // OK -- static const

    return x + y + b;
}
```

Now, you know the functions are `extern` by default, so we should be able to call `add()` from another file. You'd like to think that, wouldn't you!

But you can't! If it's just a plain `inline`, it's similar to `static`: it's only visible in that file.

Okay, so what if you throw an `extern` on there? Now we're coming full circle to when we discussed having `inline` mixed with functions with external linkage.

If both are visible, the compiler can choose which to use.

Let's do a demo of this behavior. We'll have two files, `foo.c` and `bar.c`. They'll both call `func()` which is `inline` in `foo.c` and external linkage in `bar.c`.

Here's `foo.c` with the `inline`.

```
// foo.c

#include <stdio.h>

inline char *func(void)
{
    return "foo's function";
}

int main(void)
{
    printf("foo.c: %s\n", func());

    void bar(void);
    bar();
}
```

Recall that unless we're doing an optimized build with `gcc`. `func()` will vanish and we'll get a linker error. Unless, or course, we have a version with external linkage defined elsewhere.

And we do. In `bar.c`.

```
// bar.c

#include <stdio.h>

char *func(void)
{
    return "bar's function";
}

void bar(void)
{
    printf("bar.c: %s\n", func());
}
```

So the question is, what is the output?

Seems like when we call `func()` from `foo.c`, it should print ``foo's function''. And from `bar.c`, that `func()` should print ``bar's function''.

And if I compile with `gcc` with optimizations[1] it will use inline functions, and we'll get the expected:

```
foo.c: foo's function
bar.c: bar's function
```

Great!

But if we compile in `gcc` without optimizations, it ignores the inline function and uses the external linkage `func()` from `bar.c`! And we get this:

[1] You can do this with `-O` on the command line.

```
foo.c: bar's function
bar.c: bar's function
```

In short, the rules are surprisingly complex. I give myself a good 30% chance of having described them correctly.

41.1.2 `noreturn` and `_Noreturn`

This indicates to the compiler that a particular function will not ever return to its caller, i.e. the program will exit by some mechanism before the function returns.

It allows the compiler to perhaps perform some optimizations around the function call.

It also allows you to indicate to other devs that some program logic depends on a function *not* returning.

You'll likely never need to use this, but you'll see it on some library calls like `exit()`[2] and `abort()`[3].

The built-in keyword is `_Noreturn`, but if it doesn't break your existing code, everyone would recommend including `<stdnoreturn.h>` and using the easier-to-read `noreturn` instead.

It's undefined behavior if a function specified as `noreturn` actually does return. It's computationally dishonest, see.

Here's an example of using `noreturn` correctly:

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

noreturn void foo(void) // This function should never return!
{
    printf("Happy days\n");

    exit(1);            // And it doesn't return--it exits here!
}

int main(void)
{
    foo();
}
```

If the compiler detects that a `noreturn` function could return, it might warn you, helpfully.

Replacing the `foo()` function with this:

```
noreturn void foo(void)
{
    printf("Breakin' the law\n");
}
```

gets me a warning:

```
foo.c:7:1: warning: function declared 'noreturn' should not return
```

[2] https://beej.us/guide/bgclr/html/split/stdlib.html#man-exit
[3] https://beej.us/guide/bgclr/html/split/stdlib.html#man-abort

41.2 Alignment Specifiers and Operators

Alignment[4] is all about multiples of addresses on which objects can be stored. Can you store this at any address? Or must it be a starting address that's divisible by 2? Or 8? Or 16?

If you're coding up something low-level like a memory allocator that interfaces with your OS, you might need to consider this. Most devs go their careers without using this functionality in C.

41.2.1 `alignas` and `_Alignas`

This isn't a function. Rather, it's an *alignment specifier* that you can use with a variable declaration.

The built-in specifier is `_Alignas`, but the header `<stdalign.h>` defines it as `alignas` for something better looking.

If you need your `char` to be aligned like an `int`, you can force it like this when you declare it:

```
char alignas(int) c;
```

You can also pass a constant value or expression in for the alignment. This has to be something supported by the system, but the spec stops short of dictating what values you can put in there. Small powers of 2 (1, 2, 4, 8, and 16) are generally safe bets.

```
char alignas(8) c;   // align on 8-byte boundaries
```

If you want to align at the maximum used alignment by your system, include `<stddef.h>` and use the type `max_align_t`, like so:

```
char alignas(max_align_t) c;
```

You could potentially *over-align* by specifying an alignment more than that of `max_align_t`, but whether or not such things are allowed is system dependent.

41.2.2 `alignof` and `_Alignof`

This operator will return the address multiple a particular type uses for alignment on this system. For example, maybe `char`s are aligned every 1 address, and `int`s are aligned every 4 addresses.

The built-in operator is `_Alignof`, but the header `<stdalign.h>` defines it as `alignof` if you want to look cooler.

Here's a program that will print out the alignments of a variety of different types. Again, these will vary from system to system. Note that the type `max_align_t` will give you the maximum alignment used by the system.

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

struct t {
    int a;
    char b;
    float c;
};

int main(void)
{
    printf("char       : %zu\n", alignof(char));
```

[4] https://en.wikipedia.org/wiki/Data_structure_alignment

```
    printf("short      : %zu\n", alignof(short));
    printf("int        : %zu\n", alignof(int));
    printf("long       : %zu\n", alignof(long));
    printf("long long  : %zu\n", alignof(long long));
    printf("double     : %zu\n", alignof(double));
    printf("long double: %zu\n", alignof(long double));
    printf("struct t   : %zu\n", alignof(struct t));
    printf("max_align_t: %zu\n", alignof(max_align_t));
}
```

Output on my system:

```
char       : 1
short      : 2
int        : 4
long       : 8
long long  : 8
double     : 8
long double: 16
struct t   : 16
max_align_t: 16
```

41.3 `memalignment()` Function

New in C23!

(Caveat: none of my compilers support this function yet, so the code is largely untested.)

`alignof` is great if you know the type of your data. But what if you're *woefully ignorant* of the type, and only have a pointer to the data?

How could that even happen?

Well, with our good friend the `void*`, of course. We can't pass that to `alignof`, but what if we need to know the alignment of the thing it points to?

We might want to know this if we're about to use the memory for something that has significant alignment needs. For example, atomic and floating types often behave badly if misaligned.

So with this function we can check the alignment of some data as long as we have a pointer to that data, even if it's a `void*`.

Let's do a quick test to see if a void pointer is well-aligned for use as an atomic type, and, if so, let's get a variable to use it as that type:

```
void foo(void *p)
{
    if (memalignment(p) >= alignof(atomic int)) {
        atomic int *i = p;
        do_things(i);
    } else
        puts("This pointer is no good as an atomic int\n");

...
```

I suspect you will rarely (to the point of never, likely) need to use this function unless you're doing some low-level stuff.

And there you have it. Alignment!

Index