Beej's Guide to C Programming

Brian "Beej Jorgensen" Hall

Contents

Foreward

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 entires 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 though pure C programming. Right on.

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 particularly 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 pastey 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?

Platform and Compiler

I'll try to stick to Plain Ol'-Fashioned ISO-standard C[9]. Well, for the most part. Here and there I might go crazy and start talking about POSIX[10] or something, but we'll see.

---

[1] http://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
[9] https://en.wikipedia.org/wiki/ANSI_C
[10] https://en.wikipedia.org/wiki/POSIX

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[11]. Or, if you're looking for a more Unix-like experience (recommended!), install WSL[12] and `gcc`.

Mac users will want to install XCode[13], 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 for those not in the know, a C++ compiler will compile C most code, so it'll work for the purposes of this guide.

Official Homepage

This official location of this document is http://beej.us/guide/bgc/[14]. Maybe this'll change in the future, but it's more likely that all the other guides are migrated off Chico State computers.

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.

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!

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`.

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.

Copyright and Distribution

[11] https://visualstudio.microsoft.com/vs/community/
[12] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[13] https://developer.apple.com/xcode/
[14] http://beej.us/guide/bgc/

Educators are freely encouraged to recommend or supply copies of this guide to their students.

Contact `beej@beej.us` for more information.

Hello, World!

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 used 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[15].

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[16] 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[17] or in embedded systems[18]. (Though the Rust[19] 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*[20] pointers, they're suddenly easy. But up until that moment, they're slippery eels.

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.

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[21]. Hang on!

---

[15] https://en.wikipedia.org/wiki/Assembly_language
[16] https://en.wikipedia.org/wiki/Bare_machine
[17] https://en.wikipedia.org/wiki/Operating_system
[18] https://en.wikipedia.org/wiki/Embedded_system
[19] https://en.wikipedia.org/wiki/Rust_(programming_language)
[20] https://en.wikipedia.org/wiki/Grok
[21] I know someone will fight me on that, but it's gotta be at least in the top three, right?

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.)

```
/* Hello world program */

#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");  // Actually do the work here
}
```

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 (well, technically there are more than two, but hey, let's pretend there are two and have a good laugh) 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[22], machine code[23], or whatever it's about to do. 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. Octothorpe.

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 contains preprocessor directives and function prototypes (more on that later) for common input and output needs. For our demo program, we're outputting the string "Hello, World!", so we in particular need the function prototype for the `printf()` function from this header file. 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 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.

[22] https://en.wikipedia.org/wiki/Assembly_language
[23] https://en.wikipedia.org/wiki/Machine_code

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 most everything, as you'll see.

You're passing one parameter 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 look 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") that corresponds to the *newline* character. This is the character that causing further printing to continue on 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), 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!

Compilation

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 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[24]. And it's commonly run from the command line in a terminal, but not always. You can run it from your IDE, as well.

But we'll do some command line examples here because there are too many IDEs to cover. Search the Internet for your IDE and "how to compile C" for more information.

So how do we do command line builds?

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"[25]. 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.

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, and C11. 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.

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

[25]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.

Version	Description
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 will rememeber 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 C21.

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=c99 -pedantic foo.c
```

For this book, I compile programs for C18 with all warnings set:

```
gcc -Wall -Wextra -std=c18 -pedantic foo.c
```

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.

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[26] Data is stored in this "array"[27]. 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 pointers later.
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.

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.

[26] A "byte" is an 8-bit binary number. Think of it as an integer that can only hold the values from 0 to 255, inclusive.
[27] I'm seriously oversimplifying how modern memory works, here. But the mental model works, so please forgive me.

For Unicode, things get a little different, but the basic idea is that you can start or continue the variable name with one of the characters listed in C11 §D.1, and you can continue but *not* start a variable name with any of the characters listed in C11 §D.2.

Since those are just number ranges, I'm not going to reproduce them here. If you're in an environment that supports Unicode, just try it and see if it works.

Just don't start a variable name with the "Combining Left Harpoon Above" character and you'll be fine.

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:

Type	Example	C Type
Integer	`3490`	`int`
Floating point	`3.14159`	`float`
Character (single)	`'c'`	`char`
String	`"Hello, world!"`	`char *`[28]

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[29]. They have to be initialized or else you must assume they contain some nonsense number.

> 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!

[28]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.

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

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 parameters 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:

```
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 = 3.14!
```

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

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.

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.

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.)

The `sizeof` Operator

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

This can be different on different systems, except for `char` (which is always 1 byte).

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

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

```
int a = 999;

printf("%zu", sizeof a);      // Prints 4 on my system
printf("%zu", sizeof 3.14);   // Prints 8 on my system, also
```

or you can take the `sizeof` a type (note the parentheses are required around a type name, unlike an expression):

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

We'll make use of this later on.

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!

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;
```

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.

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");
```

But more on that later.

The Comma Operator

This is an uncommonly-used way to separated 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.

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");
```

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!

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");
```

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");
}
```

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("x is 10\n");
    printf("And also 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.

The `if` 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.

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!

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");
}
```

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. 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
}
```

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.

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 cold-death of the universe.\n");
}
```

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.

```
#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:

```
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 it 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[30].

[30]This was considered such 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.

Functions

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`.

```
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`.

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 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:

```
// This function takes no parameters and returns no value:

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

```

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

Passing by Value

When you pass a value to a function, *a copy of that value* gets made in this magical mystery world known as *the stack*[31]. (The stack is just a hunk of memory somewhere that the program allocates memory on. Some of the stack is used to hold the copies of values that are passed to functions.)

For now, the important part is that *a copy* of the variable or value is being passed to the function. The practical upshot of this is that since the function is operating on a copy of the value, you can't affect the value back in the calling function directly. Like if you wanted to increment a value by one, this would NOT work:

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

int main(void)
{
    int i = 10;

    increment(i);
}
```

You might somewhat sensibly think that the value of `i` after the call would be 11, since that's what the `++` does, right? This would be incorrect. What is really happening here?

Well, when you pass `i` to the `increment()` function, a copy gets made on the stack, right? It's the copy that `increment()` works on, not the original; the original `i` is unaffected. We even gave the copy a name: `a`, right? It's right there in the parameter list of the function definition. So we increment `a`, sure enough, but what good does that do us out in `main()`? None! Ha!

That's 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*. But no fancy-schmancy name will distract you from the fact that *EVERYTHING* you pass to a function *WITHOUT EXCEPTION* is copied onto the stack 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.

But that's a story for another time.

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.

[31]Now. *technically speaking*, the C specification doesn't say anything about a stack. It's true. Your system might not use a stack deep-down for function calls. But it either does or looks like it does, and every single C programmer on the planet will know what you're talking about when you talk about "the stack". It would be just mean for me to keep you in the dark. Plus, the stack analogy is excellent for describing how recursion works.

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 and that way the function can be defined anywhere at all, as long as the *function prototype* has been declared first.

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:

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

int main(void)
{
    int i;

    i = foo();
}

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

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!

Pointers—Cower In Fear!

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!

Well, not really. But they can cause huge headaches if you don't know what you're doing when you try to mess with them.

Memory and Variables

Computer memory holds data of all kinds, right? It'll hold `float`s, `int`s, 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. You can think of it as a bunch of numbered boxes, where each box holds a byte[32] 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 ("t" is for "type", but
// it's pronounced "size tee"), which is what is returned by sizeof.
// More on size_t later.

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

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

When you have a data type 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 order, and sometimes they're not[33], but that's platform-dependent, and often taken care of for you without you needing to worry about pesky byte orderings.

So *anyway*, if we can get on with it and get a drum roll and some forboding music playing for the definition of a pointer, *a pointer is the address of some data in memory*. Imagine the classical score from 2001: A Space Odessey 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` can be `12`, a pointer can be the address of data.

[32] 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.

[33] The order that bytes come in is referred to as the *endianess* of the number. Common ones are *big endian* and *little endian*. This usually isn't something you need to worry about.

This means that all these things mean the same thing:

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

Often, we like to make a pointer to some data that we have stored in a variable, as opposed to any old random data out in memory wherever. Having a pointer to a variable is often more useful.

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 is just the *address of* the data. What operator do you suppose we'd use to find the *address of* the `int`?

Well, by a shocking suprise 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.

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[34] 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.

```c
#include <stdio.h>

int main(void)
{
    int i = 10;

    printf("The value of i is %d, and its address is %p\n", i, &i);
}
```

On my computer, this prints:

```
The value of i is 10, and its address is 0x7ffda2546fc4
```

If you're curious, that hexadecimal number is 140,727,326,896,068 in base 10. 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 literal sign with an arrow on it pointing at a thing, 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 number is, generally. We just care that it's a pointer to `i`.

Pointer Types

Well, 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.

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

Welcome back to another installment of Beej's Guide to Whatever. 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" */
}
```

Hey, so we have here a variable that is a pointer itself, and it can point to 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 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 address-of-`int` (since `i` is an `int`). But remember that "address" and "pointer" both mean the same thing! The address of a thing is pointer to that thing.

So effectively, both sides of the assignment are type pointer-to-`int` (which is the same as type "address-of-`int`", but no one says it that way).

Get it? I know is 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.

Dereferencing

Like we've said, a pointer, also known as an address, is sometimes also called a *reference*. How in the name of all that is holy can there be so many terms for exactly the same thing? I don't know the answer to that one, but these things are all equivalent, and can be used interchangeably.

The only reason I'm telling you this is so that the name of this operator will make any sense to you whatsoever. 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".)

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 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[35].

Here's a full-blown example:

```
#include <stdio.h>

int main(void)
```

[35] That's not all! It's used in `/*comments*/` and multiplication!

```
{
    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; // i (yes i!) is now 20!!

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

Remember that `p` holds the address of `i`, as you can see where we did the assignment to `p`. What the dereference operator does is tells the computer to *use the variable 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?

Passing Pointers as Parameters

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 feathered 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 parameters to functions and they'd be dutifully copied onto the stack, 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, like this:

What happens when you pass a pointer as a parameter to a function? Does a copy of the pointer get put on the stack? *You bet your sweet peas it does.* Remember how earlier I rambled on and on about how *EVERY SINGLE PARAMETER* gets copied onto the stack and the function uses a copy of the parameter? 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! Example!

```
#include <stdio.h>

void increment(int *p)  // note that it accepts a pointer to an int
{
    *p = *p + 1;  // add one to the thing p points to
}

int main(void)
{
    int i = 10;
    int *j = &i;  // note the address-of; turns it into a pointer

    printf("i is %d\n", i);        // prints "10"
    printf("i is also %d\n", *j); // prints "10"

    increment(j);
```

```

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

Ok! There are a couple things to see here…not the least of which is that the `increment()` function takes an `int*` as a parameter. 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 is an address, so we make pointers out of variables by running them through the `address-of` operator.)

The `increment()` function gets a copy of the pointer on the stack. Both the original pointer `j` (in `main()`) and the copy of that pointer `p` (in `increment()`) point to the same address, namely the one holding the value `i`. So dereferencing either will allow you to modify the original variable `i`! The function can modify a variable in another scope! Rock on!

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 and stores the result in a variable. The only way it can see that variable that is local to that calling function 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!

The `NULL` Pointer

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
```

Despite being called the billion dollar mistake by its creator[36], the `NULL` pointer is a good sentinel value[37] and general indicator that a pointer hasn't yet been initialized.

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

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?

[36] https://en.wikipedia.org/wiki/Null_pointer#History
[37] https://en.wikipedia.org/wiki/Sentinel_value

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

So a and b are both ints. 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 this line does *not* declare two pointers:

```
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[38].

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

Arrays

Luckily, C has arrays. I mean, I know it's considered a low-level language[39] 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 in C.

But only barely! 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*.

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[40].

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!

[39] These days, anyway.

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

Getting the Length of an Array

You can't. C doesn't record this information. You have to manage it separately in another variable.

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 into a function.

Array Initializers

You can initialize an array with constants ahead of time:

```
#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]);
    }
}
```

Catch: initializer values must be constant terms. Can't throw variables in there. Sorry, Illinois!

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.

```
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:

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

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

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!
```

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*[41].

Short version: don't do anything that causes undefined behavior. Ever[42].

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[43].

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

[41] 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.)

[42] 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*.

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

```c
#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
```

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.

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[44].

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

```c
#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
```

[44]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.

```

    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 is `11`, same as `a[0]`? Yessss. You're starting to get a glimpe of how arrays and pointers are related in C.

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);
}

// Same thing, but using array notation
void times3(int a[], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 3);
}

// Same thing, but using array notation with size
void times4(int a[5], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 4);
}

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

    times2(x, 5);
    times3(x, 5);
    times4(x, 5);
}
```

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 C, 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[45]). 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.

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 into a function, the function manipulates the values in that array, and those changes are visible out in the caller.

```
#include <stdio.h>

void double_array(int *a, int len)
{
    // Multiple 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!
}
```

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.

[45] 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 `struct`s 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.

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[46] 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.

Also, 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!

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

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.

Constant Strings

Before we start, let's talk about constant strings 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.

String Variables

Now that we know how to make a constant string, 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`[47]. 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!"
```

[47] It's actually type `const char*`, but we haven't talked about `const` yet.

String Variables as Arrays

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

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

// or, if we were properly lazy:

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.

String Initializers

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

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

But these two are subtly different.

This one is a pointer to a constant string (i.e. a pointer to the first character in a constant 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 constant string!
```

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 non-constant, mutable *copy* of the constant 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 constant string, don't try to change it!

Getting String Length

You can't, since C doesn't track it for you. And when I say "can't", I actually mean "can"[48]. There's a function in `<string.h>` called `strlen()` that can be used to compute the length of any string.

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

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

    printf("The string is %zu characters 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 characters 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?

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.

Of course, these days is 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.

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

- A zero-valued byte (or `NUL` character[49]) 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 constant string 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 characters 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[50]:

```
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.

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!"
}
```

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[51].

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.

[49] This is different than the `NULL` pointer, and I'll abbreviate it `NUL` when talking about the character versus `NULL` for the pointer.
[50] Later we'll learn a neater way to do with with pointer arithmetic.
[51] There's a safer function called `strncpy()` that you should probably use instead, but we'll get to that later.

```c
#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.

Structs

In C, 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[52]. You can think of a `struct` as a class with only data members, and no methods.

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;
```

And now we have an uninitialized variable `saturn`[53] 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;
saturn.speed = 175;

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

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

[53] 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.

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=172, .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).

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[54]

For those two reasons, it's far more common to pass a pointer to a `struct` to a function.

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

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

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

```
};

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, 800.00);

    // ... code continues ...
```

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 `800.0` 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 `struct`s… it doesn't work on *pointers* to `struct`s.

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 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.

The Arrow Operator

```
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!
}
```

The arrow operator helps refer to fields in pointers to `struct`s.

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 (`->`).

Copying and Returning `struct`s

Here's an easy one for you!

Just assign from one to the other!

```
struct 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". All fields are copied as-is, including pointers to things.

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.

The `FILE*` Data Type

When we do any kind of I/O in C, we do so though 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.

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[55]. 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

    char 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, 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.

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.

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

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.

We can use this to read the whole file in a loop.

[55] 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 OSX, being Unix-based, reduced this number to two.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp;
6
7      fp = fopen("hello.txt", "r");
8      char c;
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 `fgets()` 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.

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[56].

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
```

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);

```

[56]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 line.

```
    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
```

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

```c
#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);

    fclose(fp);
}
```

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
```

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);
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!

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[57].

```
#include <stdio.h>

```

[57]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.

```c
int main(void)
{
    FILE *fp;
    unsigned char bytes[] = {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), sizeof bytes, 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 as computed by `sizeof bytes`.

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[58] 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.

```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
```

[58] https://en.wikipedia.org/wiki/Hex_dump

```
37
0
88
255
12
```

Woo hoo!

`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, 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*[59] of the architecture. Some write the most significant bytes first, and some the least significant bytes first.

This means that if you write a multibyte number out straight from memory, you can't do it in a portable way[60].

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*[61], 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.

[59] https://en.wikipedia.org/wiki/Endianess
[60] And this is why I used individual bytes in my `fwrite()` and `fread()` examples, above, shrewdly.
[61] https://en.wikipedia.org/wiki/Protocol_buffers

Do yourself and everyone a favor! Serialize your binary data when you write it to a stream!

`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.

`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.

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.

`typedef` in Practice

So renaming `int` to something else isn't that exciting. Let's see where `typedef` commonly makes an appearance.

`typedef` and `struct`s

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*[62]. 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!
//          |
//          v
//      |----|
typedef struct {
    char *name;
    int leg_count, speed;
} animal;                         // <-- new name

//struct animal y;  // ERROR: this no longer works
animal z;           // This works because "animal" is an alias
```

As another example, we might find something like this:

```
typedef struct {
    int x, y;
} point;

point p = {.x=20, .y=40};

printf("%d, %d\n", p.x, p.y);  // 20, 10
```

[62] We'll talk more about these later.

`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 nee to change the `typedef`:

```
//        voila!
//      |---------|
typedef long double app_float;

// and

app_float f1, f2, f3;  // Now these are all long doubles
```

`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.

`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.

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.

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.

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.

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
```

The 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`[63] and it knows to skip that many bytes to get to the next `int` after the first one!

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
```

[63] Recall that the `sizeof` operator tells you the size in bytes of an object in memory.

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 do 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.

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 starting *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.

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[64]—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!

```c
#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!

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)
```

[64] 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.

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:

```c
#include <stdio.h>

int main(void)
{
    int a[] = {11, 22, 33, 44, 55};  // Add 999 here as a sentinel

    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.

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.:

```c
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!
```

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.

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 `s1` into the memory starting at address `s2`.

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];

    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 antelopy my_clone_antelope;

// ...

memcpy(&my_clone, &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*.

That's the power of `void*`. You can write code that doesn't care about the type and is 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 a `void*`.
2. You cannot dereference a `void*`.
3. You cannot use the arrow operator on a `void*`, since it's also a deference.
4. You cannot use array notation on a `void*`, since it's also a dereference, as well[65].

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[66].

```
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;
```

[65] Because remember that array notation is just a dereference and some pointer math, and you can't dereference a `void*`!

[66] You can also *cast* the `void*` to another type, but we haven't gotten to casts yet.

```
        // 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.

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, `float`s, `struct`s, 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
    return animal1->leg_count - animal2->leg_count;
}

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.
    //
    // This call is saying: qsort array a, which has 4 elements, and
    // each element is sizeof(struct animal) bytes big, and this is the
    // function that will compare any two elements.
    qsort(a, 4, sizeof(struct animal), compar);

    // Print them all out
    for (int i = 0; i < 4; i++) {
        printf("%d: %s\n", a[i].leg_count, a[i].name);
    }
}
```

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.

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[67].

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>`.

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.

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()`.

[67] 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.

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)
```

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
}
```

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 `char`s (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.

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(sizeof(int), 10);

// Allocate space for 10 ints with malloc(), initialized to 0:
```

```
int *q = malloc(sizeof(int) * 10);
memset(q, 0, sizeof(int) * 10);   // set to 0
```

Again, the result is the same for both except `malloc()` doesn't zero the memory by default.

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).

Let's allocate an array of 20 `float`s, and then change our mind and make it an array of 40.

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

int main(void)
{
    // Allocate space for 20 floats
    float *p = malloc(sizeof *p * 20);

    // 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
    p = realloc(p, sizeof *p * 40);

    // And assign the new elements values in the range 1.0-2.0
    for (int i = 20; i < 40; i++)
        p[i] = 1.0 + (i - 20) / 20.0;

    // Print all values 0.0-2.0 in the 40 elements:
    for (int i = 0; i < 40; i++)
        printf("%f\n", p[i]);

    // Free the space
    free(p);
}
```

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.

`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.

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 `struct`s. 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[68]

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

```

[68] http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_460

```
    // 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 alignment, size_t size)
{
    char *new_ptr = aligned_alloc(alignment, size);

    if (new_ptr == NULL)
        return NULL;

    if (ptr != NULL)
        memcpy(new_ptr, ptr, size);

    return new_ptr;
}
```

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.

Scope

Scope is all about what variables are visible in what contexts.

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[69] in the sky.

An example with nested scope:

```
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
}
```

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.

```
#include <stdio.h>

int main(void)
{
    int i = 0;
```

[69] https://en.wikipedia.org/wiki/Bit_bucket

```

    printf("%d\n", i);     // OK: "0"

    //printf("%d\n", j);   // ILLEGAL--can't use j before it's defined

    int j = 5;

    printf("%d %d\n", i, j);   // OK: "0 5"
}
```

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.

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 at long as you're running in the inner scope. That is, it *hides* the one at outer scope for th 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.

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:

```
#include <stdio.h>

int shared = 10;    // File scope! Visible to the whole file after this!

void func1(void)
{
    shared += 100;  // Now shared holds 110
}
```

```c

void func2(void)
{
    printf("%d\n", shared);  // Prints "10"
}

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.

`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);
    }
}
```

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.

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!

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*[70]. 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[71] to represent unsigned numbers, I have the following limits on integer range:

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.

[70] "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.

[71] https://en.wikipedia.org/wiki/Two%27s_complement

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."[72] 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[73].

So, assuming those constraints, we can finally figure our ranges:

`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`!

[72]The industry term for a sequence of exactly, indisputably 8 bits is an *octet*.

[73]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}.

```c
#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[74] 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:

```c
#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[75]. One of the most popular character sets today is Unicode[76] (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.

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.

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.

```c
// These two lines are equivalent:
long long int x;
long long x;
```

[74] https://en.wikipedia.org/wiki/ASCII
[75] https://en.wikipedia.org/wiki/List_of_information_system_character_sets
[76] https://en.wikipedia.org/wiki/Unicode

```
// 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[77]
`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	44294967295
`unsigned long long`	8	0	9223372036854775807

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[78]) 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[79]
`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
`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`

[77] Depends on if a `char` defaults to `signed char` or `unsigned char`
[78] https://en.wikipedia.org/wiki/Signed_number_representations#Signed_magnitude_representation
[79] My `char` is signed.

Type	Min Macro	Max Macro
`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`.

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:
>
> $$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \le e \le e_{max}$$

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[80] 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:

$12345 \times 10^{-3} = 12.345$

$12345 \times 10^{4} = 123450000$

[80] https://en.wikipedia.org/wiki/IEEE_754

$12345 \times 10^0 = 12345$

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.

How Many Decimal Digits?

The million dollar question is, "How many significant decimal digits can I store in a given floating point type before the floating point precision runs out?"

But it's not quite so easy to answer. So we'll do it in two ways.

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—some numbers will come back correctly with more digits. But 6 is definitely coming back.)

For example, printing out `float`s 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 `float`s that both hold numbers that have `FLT_DIG` significant decimal digits[81]. 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.

```c
#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 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?

Converting to Decimal and Back

But storing a base 10 number in a floating point number is only half the story.

What about when you print out a floating point number? How many digits can you print?

You might think it would be the same as the number you can store, but it's not[82]!

But recall that you might have more decimal digits than `FLT_DIG` encoded correctly in the number. In order to make sure you're printed them all out, you can Of course, if you store the number `3.14f` in a `float`, you can't expect to print out more than 2 decimal places and get sensible results. But `FLT_DIG` (if 6) says that you can't store more digits than `3.14159f` and be sure of getting it stored successfully.

[81] 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.

[82] Or at least, it's probably not—if you store floating point numbers in base 2.

But what if you did some math on a floating point number? Can you get more precision?

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 the how C decides what type the constant is, and how to force it to choose a specific type.

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)
```

A Note on Binary

An unofficial extension[83] 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.

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.

```
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;
```

[83] 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.

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`

`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.

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.

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*[84]. In C, these are written using "E notation", so these are equivalent:

Scientific Notation	E notation
$1.2345 \times 10^{-3} = 12.345$	`1.2345e-3`
$1.2345 \times 10^4 = 123450000$	`1.2345e+4`

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.

[84] https://en.wikipedia.org/wiki/Scientific_notation

```
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
```

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
```

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.

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.

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*)[85]

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
}
```

If we wanted to convert a `double`, we'd use `%lf`. Or a `long double`, `%Lf`.

[85] 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.

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[86], 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.

[86] 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[87].

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.

Numeric Conversions

Boolean

If you convert a zero to `bool`, the result is `0`. Otherwise it's `1`.

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[88].

[87] We have to pass a pointer to `badchar` into `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`.

[88] 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`.

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[89]

Integer and Floating Point Conversions

If a floating point type is converted to an integer type, the fractional part is discarded with prejudice[90].

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.

Implicit Conversions

These are conversions the compiler does automatically for you when you mix and match types.

The Integer Promotions

In a number of places, if a `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[91], unary `+` and `-` operators, or when passing values to functions without prototypes[92].

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
float y = 12 * 2;  // Mixing float and int
```

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.

[89]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`.

[90]Not really—it's just discarded regularly.

[91]Functions with a variable number of arguments.

[92]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.

`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*
```

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*.

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)
{
    return *((const int*)elem2) - *((const int*)elem1);
}
```

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;
```

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.

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*.

Type Qualifiers

These are going to allow you to declare constant values, and also to give the compiler optimization hints that it can use.

`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"
}
```

`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!
```

`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
```

`restrict`

TLDR: you never have to use this and you can ignore it every time you see it.

`restrict` is a hint to the compiler that a particular piece of memory will only be accessed by one pointer and never another. If a developer declares a pointer to be `restrict` and then accesses the object it points to in another way, 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 example, 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 an try passing in pointers to the same thing.

```
void swap(int *restrict a, int *restrict b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}

int main(void)
{
    int x = 10, y = 20;

    swap(&x, &y);  // OK! "a" and "b", above, point to different things

    swap(&x, &x);  // Undefined behavior! "a" and "b" point to the same thing
}
```

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 its 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, the restriction covers the entire array.

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.

`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;
```

`_Atomic`

This is an optional C feature that we'll talk about another time.

Type Specifiers

Type specifiers are similar to type quantifiers. They give the compiler more information about the type of a variable.

`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!

`static`

This keyword has two meanings, depending on if the variable is file scope or block scope.

Let's start with block scope.

`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.

`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.

`extern`

The `extern` type 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.

`register`

Barely anyone uses this anymore.

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
```

A bit of backstory, here: deep inside the CPU are little dedicated "variables" called *registers*[93]. 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.

In short, you probably don't want to even bother with `register`, and just let the compiler do what it thinks is best.

[93]https://en.wikipedia.org/wiki/Processor_register

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.

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`:

```c
// File bar.c

int add(int x, int y)
{
    return x + y;
}
```

And the file `foo.c` with main in it:

```c
// 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 warning:

```
warning: implicit declaration of function 'add'
```

But if we ignore that (which really we should never do—always get your code to build with zero warnings!) and try to run it:

```
./foo
5
```

Indeed, we get the result of $2 + 3$! Yay!

So… about that warning. Let's 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:

```c
// 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 warning!

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:

```c
// 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):

```c
// 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.

We're almost there! There's just one more piece of boilerplate we have to add.

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 `#include`d 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 `#include`s 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 `#include`s, 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.

`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` type specifiers.

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[94] 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.)

[94] https://en.wikipedia.org/wiki/Boids

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.

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`[95].

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:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

[95]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.

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:

```
#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`[96] (pronounced *arg-c* and *arg-v*)?

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!

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

int main(int argc, char **argv)
{
    int total = 0;

    for (int i = 0; i < argc; i++) {
        int value = atoi(argv[i]);  // Use strtol() for better error handling

        total += value;
    }

```

[96]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!

```
    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.

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.

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:

```c
#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; *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.

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[97].

  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"`:

  ```c
  strcpy(argv[0], "Hi!  ");
  ```

  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.

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

[97] `ps`, Process Status, is a Unix command to see what processes are running at the moment.

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[98] 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[99] 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.

```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
    }

```

[98] https://en.wikipedia.org/wiki/Inception
[99] https://en.wikipedia.org/wiki/Shell_(computing)

```
    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[100].

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.

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 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.

[100] In Windows `cmd.exe`, type `echo %errorlevel%`. In PowerShell, type `$LastExitCode`.

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 except 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[101].

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.

```c
#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:

```
$ ./foo
Cannot find the FROTZ environment variable
```

which makes since, since I haven't set it yet.

In bash, I can set it to something with[102]:

```
$ export FROTZ="C is awesome!"
```

Then if I run it, I get:

```
$ ./foo
Value: C is awesome!
```

[101] If you need a numeric value, convert the string with something like `atoi()` or `strtol()`.
[102] In Windows CMD.EXE, use `set FROTZ=value`. In PowerShell, use `$Env:FROTZ=value`.

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.

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()`.

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!

`#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.

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:

```
#include <stdio.h>

int main(void)
{
    printf("%s, %f\n", "Hello, world", 3.14159);
}
```

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 that a constant can be used, e.g. in `switch` `case`s.

It can also be used to replace or modify keywords, a place a `const` won't work at all, though this practice should be used sparingly.

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.

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 get compiled or removed depending on the condition. This is in contrast to a standard `if` statement that gets evaluated while the program is running.

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 read to output different things based on whether or not something was defined:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#endif

#ifndef EXTRA_HAPPY
    printf("I'm just regular\n");
#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 `#include`s. 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.

`#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
```

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[103].

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
```

You might have noticed that there's no `#elifdef` or `#elifndef` directives. How can we get the same effect with `#if`? That is, what if I wanted this:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // ERROR: Not supported by standard C
    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.

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  // ERROR: Not supported by standard C
    x = 3;
#endif
```

can be replaced with:

```
#if defined FOO
    x = 2;
#elif defined BAR
    x = 3;
#endif
```

[103] You can't always just wrap the code in `/* */` comments because those won't nest.

Losing a Macro: `#undef`

If you've defined something but you don't need it any longer, you can undefine it with `#undef`.

```c
#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
}
```

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.

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[104]
`__STDC__`	Defined with `1` if this is a standard C compiler
`__STDC_HOSTED__`	This will be `1` if the compiler is a *hosted implementation*[105], otherwise `0`
`__STDC_VERSION__`	This version of C, a constant `long int` in the form `yyyymmL`, e.g. `201710L`

Let's put these together.

```c
#include <stdio.h>

int main(void)
```

[104] 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.

[105] 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*.

```c
{
    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.

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[106]
__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

[106] 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.

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.

Macros with One Argument

Let's start with a simple one that squares a number:

```
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:

```
#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.

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):

$$2 \times -0.563508^2 + 10 \times -0.563508 + 5 \approx 0.000003$$

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:

```
1  #include <stdio.h>
2
3  // 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"
```

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.

```
#include <stdio.h>

#define PRINT_INT_VAL(x) printf("%s = %d\n", #x, x)

int main(void)
{
    int a = 5;
```

```
8
9       PRINT_INT_VAL(a);  // prints "a = 5"
10  }
```

On line 9, we get the following macro replacement:

```
    printf("%s = %d\n", "a", 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
```

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) { \
    int product = (a) * (b); \
    for (int i = 0; i < product; i++) { \
        printf("%d\n", i); \
    } \
}

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.
- Though not strictly necessary, I wrapped the whole thing in curly braces. This did two things:
  1. Made it look nice.
  2. Made a new block scope for my `product` variable so it wouldn't conflict with any other existing variables at the outer block scope.

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.

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.

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[107] 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.

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:
>
> $(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$
>
> $(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$
>
> $|x + iy| = \sqrt{x^2 + y^2}$
>
> where the programmer can determine they are safe.

[107] https://www.openmp.org/

`_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)
```

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"
```

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.

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.

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 `struct`s. 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.

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 node 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
```

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 `struct`s 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:

```c
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:

```c
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`.

```c
// 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:

```c
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!

```c
// 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[108]:

[108] Technically we say that it has an *incomplete type*.

```
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:

```
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;
}
```

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[109].

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
```

[109] Though some compilers have options to force this to occur—search for `__attribute__((packed))` to see how to do this with GCC.

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.

`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>`.

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.

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.

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;
};
```

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;
};
```

Put all your bitfields together to get the compiler to combine them.

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.

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[110].

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;
};
```

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`.

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."

[110] Assuming 8-bit `char`s, i.e. `CHAR_BIT == 8`.

Unions

These are basically just like `struct`s, 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 `struct`s, 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. If you try to read from a field that was not the last one written to, the behavior is unspecified.

Let's take that crazy `union` and first store an `int` in it, then a `float`. Then we'll print out the `int` again to see what's in there—even though, since it wasn't the last value we stored, the result is unspecified.

```
#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;

    x.a = 12;
    printf("%d\n", x.a);  // OK--x.a was the last thing we stored into

    x.g = 3.141592;
    printf("%f\n", x.g);  // OK--x.g was the last thing we stored into

    printf("%d\n", x.a);  // Unspecified behavior!
}
```

On my machine, this prints:

```
12
3.141592
1078530008
```

Probably deep down the decimal value `1078530008` is probably the same pattern of bits as `3.141592`, but the spec makes no guarantees about this.

Pointers to `union`s

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

    x.g = 3.141592;
    printf("%f\n", x.g);           // 3.141592
    printf("%f\n", *foo_float_p);  // 3.141592, again
}
```

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.

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!

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.

Frequently-used Escapes

In my humble opinion, these escape characters make up 99.2%[111] of all escapes.

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

[111] I just made up that number, but it's probably not far off

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", '\'');             // '
```

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.

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.)

```
1  #include <stdio.h>
2  #include <unistd.h>  // Non-standard Unix-likes only for sleep()
3
4  int main(void)
5  {
6      for (int i = 10; i >= 0; i--) {
7          printf("\rT minus %d second%s... \b", i, i != 1? "s": "");
8
9          fflush(stdout);  // Force output to update
10
11         sleep(1);        // Delay 1 second
12     }
13
14     printf("\rLiftoff!             \n");
15 }
```

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*.

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.

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?

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.

Behavior of enum

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 2
```

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
}
```

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.

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 `#include`d at file scope.

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.

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, `enum`s are a great way to write nice, scoped, typed, clean code.

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.

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[112].

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:

```
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 
8     printf("%d\n", *p);  // 3490
9 }
```

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?

[112]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.

```
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[113] 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`[114] and then goes up by 8 from `p` to `q`. On my system, all pointers are 8 bytes.

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:

[113] You're very likely to get different numbers on yours.
[114] 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.

- 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`[115]. 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:

```c
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 *****
```

Pointer Pointers and `const`

If you recall, declaring a pointer like this:

```c
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?

[115] Even if `E` is `NULL`, it turns out, weirdly.

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:

```
int main(void)
{
    int x = 3490;
    int *const p = &x;
    int **q = &p;
}
```

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

That is, we're saying that `q` is type `int **`, and if you dereference that, the rightmost `*` in the type goes away. So after the dereference, we have type `int *`.

And we're assigning `&p` into it which is *a pointer to* an `int *const`, or, in other words, `int *const *`.

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.

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 `char`s). 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()` 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[116].

```
#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]);
    }
}
```

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
```

[116] Your C compiler is not required to add padding bytes, and the values of any padding bytes that are added are indeterminate.

```
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[117].

So that's a little peek under the hood at the bytes of a more complex entity in memory.

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.

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 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>`[118].

Additionally, if you feel like being signed, you can use `intptr_t` to the same effect.

[117] 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.
[118] It's an optional feature, so it might not be there—but it probably is.

Pointer Differences

As you know from the section on pointer arithmetic, you can subtract one pointer from another[119] 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
```

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.

```
// 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.)

---

[119] Assuming they point to the same array object.

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);

    printf("%d\n", result);
}

int main(void)
{
    print_math(add, 5, 7);   // 12
    print_math(mult, 5, 7);  // 35
}
```

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[120].

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()`.

[120]The Go Programming Language drew its type declaration syntax inspiration from the opposite of what C does.

Bitwise Operations

These numeric operations effectively allow you to manipulate individual bits in variables, fitting since C is such a low-level langauge[121].

If you're not familiar with bitwise operations, Wikipedia has a good bitwise article[122].

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`

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.

Operation	Operator	Example
Shift left	`<<`	`a = b << c`
Shift right	`>>`	`a = b >> c`

[121] 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.

[122] https://en.wikipedia.org/wiki/Bitwise_operation

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.)

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!

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!

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;
    }

    va_end(va);  // All done

    return total;
}

int main(void)
```

```
{
    printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
}
```

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.

`va_list` Functionality

What is that `va_list` variable we're using up there? It's an opaque variable[123] 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[124].

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 on won't be started, either. If you've consumed 5 variables with `va_arg()` so far, the copy will also reflect that.

`va_copy()` can be useful if you need to scan ahead through the arguments but need to also remember your current place.

[123] 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.

[124] 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.

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!

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:

```
set_locale(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:

```
set_locale(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."

Getting the Monetary Locale Settings

Because moving green pieces of paper around promises to be the key to happiness[125], 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.

"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[126].

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`.
`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`.

[125] "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

[126] Remember that `char` is just a byte-sized integer.

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 `char`s. 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.

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.

Value	Description
`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.

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
```

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[127].
`LC_MONETARY`	Controls the values returned by `localeconv()`.

Macro	Description
`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.

[127] Except for `isdigit()` and `isxdigit()`.

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 take 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!

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, cuniform, chess symbols, emojis… just about everything, really! And more is being added all the time!

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.

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[128]. 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[129] 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[130].

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

Look in there for the patterns. Note that UTF-16BE and UTF-32BE are simply the code point represented

[128]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!

[129]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.]

[130]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.

directly as 16- and 32-bit values[131].

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[132] 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[133].

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.

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.

Notably, it's a pain (though possible with escape sequences) to enter Unicode characters using only the basic character set.

[131] Again, this is only true in UTF-16 for characters that fit in two bytes.
[132] https://en.wikipedia.org/wiki/UTF-8
[133] https://www.youtube.com/watch?v=MijmeoH9LT4

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[134].

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 (`` ` ``).

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).

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.

[134] 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.

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. (Use `mbstowcs()` with a `NULL` first argument to get the number of characters in a multibyte string.)
- 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[135]. 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 `char`s 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…

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!

Multibyte Characters

First of all, I want to potentially change your thinking about what a string (array of `char`s) is. These are *multibyte strings* made up of *multibyte characters*.

That's right—your run-of-the-mill string of characters is multibyte.

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 multibyte.

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:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

But here we're getting into some weirdness, because check this out:

[135] With a format specifier like `"%s.12"`, for example.

```
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, 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');
```

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.

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 function you can use to deal with `wchar_t` strings (like getting the length, etc.) without caring about the encoding.

Using Wide Characters and `wchar_t`

Time for a new type: `wchar_t`. This is the main wide character typre. 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 are 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?

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. And, in fact, it has 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
```

And, of course, if you want to convert the other way, it's `wcstombs()`.

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 an `wcslen()` for wide character strings.)

`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.

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()`.

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.

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`.

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.

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()`[136].

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.

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.

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.

[136] `wcscoll()` is the same as `wcsxfrm()` followed by `wcscmp()`.

Length/Misc Function	Description
`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.

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>
```

```c
#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.
    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);

    // 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);
}
```

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:

```c
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.

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.

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.

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".)

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 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
```

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[137].

Conversion Function	Description
`mbrtoc16()`	Convert a multibyte character to a `char16_t` character.
`mbrtoc32()`	Convert a multibyte character to a `char32_t` character.
`c16rtomb()`	Convert a `char16_t` character to a multibyte character.
`c32rtomb()`	Convert a `char32_t` character to a multibyte character.

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[138]—Internationalization Conversion, a common POSIX-standard API available on the major platforms.
- ICU[139]—International Components for Unicode. At least one blogger found this easy to use.

If you have more noteworthy libraries, let me know.

[137] Ish—things get funky with multi-`char16_t` UTF-16 encodings.
[138] https://en.wikipedia.org/wiki/Iconv
[139] http://site.icu-project.org/

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>`.

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.

Returning From `main()`

This is one we've seen already a bunch of times, but here's a quick refresher.

- You can return an exit status from `main()` with a `return` statement. `main()` is the only function with this special behavior.
- 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`.

`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.

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:

```c
#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!
```

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()`.

```c
#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.

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*.

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[140] as the program terminated.

[140] https://en.wikipedia.org/wiki/Core_dump

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[141].

With that out of the way, what are signals?

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.

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.
- `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.)

[141] Apparently it doesn't do Unix-style signals at all deep down, and they're simulated for console apps.

```
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...\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.

Writing Signal Handlers

I mentioned you could also write a handler function that gets called with 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 interesting 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 first 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.

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.

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[142].) 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`[143].

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.

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);

```

[142] Confusingly, `sig_atomic_t` predates the lock-free atomics and is not the same thing.

[143] If `sig_action_t` is signed, the range will be at least `-127` to `127`. If unsigned, at least `0` to `255`.

```
    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. We can do some ridiculous contortions so that we're only assigning into the values and manage to avoid undefined behavior.

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

void sigint_handler_2(int signum)
{
    (void)signum;                       // Unused variable warning
    signal(SIGINT, SIG_DFL);            // Reset signal handler
}

void sigint_handler_1(int signum)
{
    (void)signum;                       // Unused variable warning
    signal(SIGINT, sigint_handler_2);  // Set to second handler
}

int main(void)
{
    signal(SIGINT, sigint_handler_1);

    printf("Hit ^C twice to exit.\n");

    while(1);
}
```

That's pretty ugly, all right. 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), but we're getting into zanyland here.

This is why at the beginning, I was suggesting checking out your OS's built-in signal system as a probably-superior alternative.

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.

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
```

Let's dive in first with an example, and then we'll look for the devil in the details.

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);
    scanf(" %d", &n);

    int v[n];

    for (int i = 0; i < n; i++)
        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[144].
- 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.

`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:

```
#include <stdio.h>

int main(void)
{
    int n = 5;
    int v[n];

    int *p = v;

    *(p+2) = 12;
    printf("%d\n", v[2]);  // 12

    p[3] = 34;
    printf("%d\n", v[3]);  // 34
}
```

[144]This is due to how VLAs are typically allocated on the stack, where as `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.

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.

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 add(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", add(5, x));
    printf("%d\n", add(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 add(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.

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 into 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);
}
```

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);
}
```

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, through: if your dimensions mismatch, you're going to have some undefined behavior going on, likely.

`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;

```

```c
    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.

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.

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[145].

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[146].

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!
```

`goto` sends execution jumping to the specified label, skipping everything in between.

You can jump forward or backward with `goto`.

[145] https://en.wikipedia.org/wiki/Goto#Criticism

[146] I'd like to point out that using `goto` in all these cases is avoidable. You can use variables and loops intead. It's just that some people think `goto` produces the *best* code in those circumstances.

```
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.

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.

But what if we wanted it to continue in the outer loop instead? We could rewrite like this to accomplish it with `goto`.

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            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).

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.

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");
```

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.

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.

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: 'a' 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.

`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.

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!

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, `struct`s, 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[147]:

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

printf("%d\n", p[1]);  // 2
```

So let's take a look at a more useful example.

Passing Unnamed Objects to Functions

Let's say we have a function to sum an array of `int`s:

[147] Which isn't quite the same, since it's an array, not a pointer to an `int`.

```
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 second argument:

```
//                   p[]         count
//             |-----------------|   |
int s = sum((int []){1, 2, 3, 4}, 4));
```

Pretty slick!

Unnamed `struct`s

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`:

"' {.c .numberLines .startFrom="14"} //struct coord t = {.x=10, .y=20};

~~~~
print_coord((struct coord){.x=10, .y=20});   // prints "10, 20"

Still works!

### 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

``` {.c .numberLines}
#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"
}
~~~~

Easy as that.

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:

```c
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.

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.

Generic Selections

This is an expression that allows you 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[148]) 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.

```
#include <stdio.h>

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

[148] A variable used here *is* an expression.

```
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 look 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) { \
    char fmt[512]; \
    snprintf(fmt, sizeof fmt, #x " = %s\n", FMTSPEC(x)); \
    printf(fmt, (x)); \
}

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.

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.

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.

`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 cone optimization, safe in the knowledge that you, the programmer, will always do the right thing.

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:

```c
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.)

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.

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>`.

```c
#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 into `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.

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[149].

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.

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`.

Technically, they only have to be `volatile` if they change between the time `setjmp()` is called and `longjmp()` is called[150].

For example, if we run this code:

[149] Both "stack pointer" and "program counter" are related to the underlying architecture and C implementation, and are not part of the spec.

[150] 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.

```
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.

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 (including values in local variables when `setjmp()` was called)

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.

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)) {...}
  ```

  ```
  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);
```

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.

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.

'longjmp() and Variable Length Arrays

If you are in scope of a VLA and `longjmp()` out there, the memory allocated to the VAL could leak[151].

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.

[151] That is, remain allocated until the program ends with no way to free it.

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 `struct`s `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?

Use Case: Self-Referential Structures

I only know of one real use case: forward references to `struct`s or `union`s 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 `union`s, 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!

```
1 struct node {
2     int val;
3     struct node *next;  // struct node is incomplete, but that's OK!
4 };
```

Even though the `struct node` is incomplete on line 3, we can still declare a pointer to one[152].

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.

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.

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.

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

[152]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.

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 specific all the `.c` files to the compiler, but not the `.h` files, e.g.:

```
gcc -o foo foo.c bar.c
```

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…

Complex Numbers

A tiny primer on Complex numbers[153] 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.

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`.

You also get some types for imaginary numbers if you implementation is IEC 60559-compliant:

```
_Imaginary
imaginary
```

[153] https://en.wikipedia.org/wiki/Complex_number

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?

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:

```c
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!

```c
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;
```

For 5_2i and $10 + 3i$, respectively.

Constructing, Deconstructing, and Printing

We're getting there…

We've already seen one way to write a complex number:

```c
double complex x = 5 + 2*I;
```

There's also no problem using other floating point numbers to build it:

```c
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:

```c
double complex x = CMPLX(5, 2);
```

As far as I can tell in my research, these are *almost* equivalent:

```c
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*[154] 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()`.

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));

    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));
```

[154] 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));
}
```

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[155].

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.

Trigonometry Functions

Function	Description
`ccos()`	Cosine
`csin()`	Sine
`ctan()`	Tangent
`cacos()`	Arc cosine
`casin()`	Arc sine

[155]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
`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

Exponential and Logarithmic Functions

Function	Description
`cexp()`	Base-e exponential
`clog()`	Natural (base-e) logarithm

Power and Absolute Value Functions

Function	Description
`cabs()`	Absolute value
`cpow()`	Power
`csqrt()`	Square root

Manipulation Functions

Function	Description
`creal()`	Return real part
`cimag()`	Return imaginary part
`CMPLX()`	Construct a complex number
`carg()`	Argument/phase angle
`conj()`	Conjugate[156]
`cproj()`	Projection on Riemann sphere

[156] This is the only one that doesn't begin with an extra leading `c`, strangely.

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.

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`)[157]

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
```

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
```

[157] 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.

```
int_fast16_t      uint_fast16_t
int_fast32_t      uint_fast32_t
int_fast64_t      uint_fast64_t
```

There might be others, 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[158]. 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 might be defined, but they're optional.

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.

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);
```

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
```

[158] 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.

```
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.

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 `PRId16`.

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>
```

```c

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>`.

`<stdio.h>` Standard I/O Library

The most basic of all libraries in the whole of the standard C library is the standard I/O library. It's used for reading from and writing to files. I can see you're very excited about this.

So I'll continue. It's also used for readingand writing to the console, as we've already often seen with the `printf()` function.

(A little secret here—many many things in various operating systems are secretly files deep down, and the console is no exception. "*Everything in Unix is a file!*" `:-)`)

You'll probably want some prototypes of the functions you can use, right? To get your grubby little mittens on those, you'll want to include `stdio.h`.

Anyway, so we can do all kinds of cool stuff in terms of file I/O. LIE DETECTED. Ok, ok. We can do all kinds of stuff in terms of file I/O. Basically, the strategy is this:

1. Use `fopen()` to get a pointer to a file structure of type `FILE*`. This pointer is what you'll be passing to many of the other file I/O calls.
2. Use some of the other file calls, like `fscanf()`, `fgets()`, `fprintf()`, or etc. using the `FILE*` returned from `fopen()`.
3. When done, call `fclose()` with the `FILE*`. This let's the operating system know that you're truly done with the file, no take-backs.

What's in the `FILE*`? Well, as you might guess, it points to a `struct` that contains all kinds of information about the current read and write position in the file, how the file was opened, and other stuff like that. But, honestly, who cares. No one, that's who. The `FILE` structure is *opaque* to you as a programmer; that is, you don't need to know what's in it, and you don't even *want* to know what's in it. You just pass it to the other standard I/O functions and they know what to do.

This is actually pretty important: try to not muck around in the `FILE` structure. It's not even the same from system to system, and you'll end up writing some really non-portable code.

One more thing to mention about the standard I/O library: a lot of the functions that operate on files use an "f" prefix on the function name. The same function that is operating on the console will leave the "f" off. For instance, if you want to print to the console, you use `printf()`, but if you want to print to a file, use `fprintf()`, see?

Wait a moment! If writing to the console is, deep down, just like writing to a file, since everything in Unix is a file, why are there two functions? Answer: it's more convenient. But, more importantly, is there a `FILE*` associated with the console that you can use? Answer: YES!

There are, in fact, *three* (count 'em!) special `FILE*`s you have at your disposal merely for just including `stdio.h`. There is one for input, and two for output.

That hardly seems fair—why does output get two files, and input only get one?

That's jumping the gun a bit—let's just look at them:

Stream	Description
`stdin`	Input from the console.
`stdout`	Output to the console.
`stderr`	Output to the console on the error file stream.

So standard input (`stdin`) is by default just what you type at the keyboard. You can use that in `fscanf()` if you want, just like this:

```
/* this line: */
scanf("%d", &x);

/* is just like this line: */
fscanf(stdin, "%d", &x);
```

And `stdout` works the same way:

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n"); /* same as previous line! */
```

So what is this `stderr` thing? What happens when you output to that? Well, generally it goes to the console just like `stdout`, but people use it for error messages, specifically. Why? On many systems you can redirect the output from the program into a file from the command line…and sometimes you're interested in getting just the error output. So if the program is good and writes all its errors to `stderr`, a user can redirect just `stderr` into a file, and just see that. It's just a nice thing you, as a programmer, can do.

`remove()`

Delete a file

Synopsis

```
#include <stdio.h>

int remove(const char *filename);
```

Description

Removes the specified file from the filesystem. It just deletes it. Nothing magical. Simply call this function and sacrifice a small chicken and the requested file will be deleted.

Return Value

Returns zero on success, and `-1` on error, setting `errno`.

Example

```
char *filename = "/home/beej/evidence.txt";

remove(filename);
remove("/disks/d/Windows/system.ini");
```

See Also

`rename()`

`rename()`

Renames a file and optionally moves it to a new location

Synopsis

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Description

Renames the file `old` to name `new`. Use this function if you're tired of the old name of the file, and you are ready for a change. Sometimes simply renaming your files makes them feel new again, and could save you money over just getting all new files!

One other cool thing you can do with this function is actually move a file from one directory to another by specifying a different path for the new name.

Return Value

Returns zero on success, and `-1` on error, setting `errno`.

Example

```
rename("foo", "bar"); // changes the name of the file "foo" to "bar"

// the following moves the file "evidence.txt" from "/tmp" to
// "/home/beej", and also renames it to "nothing.txt":
rename("/tmp/evidence.txt", "/home/beej/nothing.txt");
```

See Also

`remove()`

`tmpfile()`

Create a temporary file

Synopsis

```
#include <stdio.h>

FILE *tmpfile(void);
```

Description

This is a nifty little function that will create and open a temporary file for you, and will return a `FILE*` to it that you can use. The file is opened with mode "`r+b`", so it's suitable for reading, writing, and binary data.

By using a little magic, the temp file is automatically deleted when it is `close()`'d or when your program exits. (Specifically, `tmpfile()` unlinks the file right after it opens it. If you don't know what that means, it won't affect your `tmpfile()` skill, but hey, be curious! It's for your own good!)

Return Value

This function returns an open `FILE*` on success, or `NULL` on failure.

Example

```
#include <stdio.h>

int main(void)
{
    FILE *temp;
    char s[128];

    temp = tmpfile();

    fprintf(temp, "What is the frequency, Alexander?\n");

    rewind(temp); // back to the beginning

    fscanf(temp, "%s", s); // read it back out

    fclose(temp); // close (and magically delete)
}
```

See Also

`fopen()`, `fclose()`, `tmpnam()`

`tmpnam()`

Generate a unique name for a temporary file

Synopsis

```
#include <stdio.h>

char *tmpnam(char *s);
```

Description

This function takes a good hard look at the existing files on your system, and comes up with a unique name for a new file that is suitable for temporary file usage.

Let's say you have a program that needs to store off some data for a short time so you create a temporary file for the data, to be deleted when the program is done running. Now imagine that you called this file `foo.txt`. This is all well and good, except what if a user already has a file called `foo.txt` in the directory that you ran your program from? You'd overwrite their file, and they'd be unhappy and stalk you forever. And you wouldn't want that, now would you?

Ok, so you get wise, and you decide to put the file in `/tmp` so that it won't overwrite any important content. But wait! What if some other user is running your program at the same time and they both want to use that filename? Or what if some other program has already created that file?

See, all of these scary problems can be completely avoided if you just use `tmpnam()` to get a safe-ready-to-use filename.

So how do you use it? There are two amazing ways. One, you can declare an array (or `malloc()` it—whatever) that is big enough to hold the temporary file name. How big is that? Fortunately there has been a macro defined for you, `L_tmpnam`, which is how big the array must be.

And the second way: just pass `NULL` for the filename. `tmpnam()` will store the temporary name in a static array and return a pointer to that. Subsequent calls with a `NULL` argument will overwrite the static array, so be sure you're done using it before you call `tmpnam()` again.

Again, this function just makes a file name for you. It's up to you to later `fopen()` the file and use it.

One more note: some compilers warn against using `tmpnam()` since some systems have better functions (like the Unix function `mkstemp()`.) You might want to check your local documentation to see if there's a better option. Linux documentation goes so far as to say, "Never use this function. Use `mkstemp()` instead."

I, however, am going to be a jerk and not talk about `mkstemp()` because it's not in the standard I'm writing about. Nyaah.

The macro `TMP_MAX` holds the number of unique filenames that can be generated by `tmpnam()`. Ironically, it is the *minimum* number of such filenames.

Return Value

Returns a pointer to the temporary file name. This is either a pointer to the string you passed in, or a pointer to internal static storage if you passed in `NULL`. On error (like it can't find any temporary name that is unique), `tmpnam()` returns `NULL`.

Example

```
char filename[L_tmpnam];
char *another_filename;

```

```
if (tmpnam(filename) != NULL)
    printf("We got a temp file named: \"%s\"\n", filename);
else
    printf("Something went wrong, and we got nothing!\n");

another_filename = tmpnam(NULL);
printf("We got another temp file named: \"%s\"\n", another_filename);
printf("And we didn't error check it because we're too lazy!\n");
```

On my Linux system, this generates the following output:

```
We got a temp file named: "/tmp/filew9PMuZ"
We got another temp file named: "/tmp/fileOwrgPO"
And we didn't error check it because we're too lazy!
```

See Also

`fopen()`, `tmpfile()`

`fclose()`

The opposite of `fopen()`—closes a file when you're done with it so that it frees system resources.

Synopsis

```
#include <stdio.h>

int fclose(FILE *stream);
```

Description

When you open a file, the system sets aside some resources to maintain information about that open file. Usually it can only open so many files at once. In any case, the Right Thing to do is to close your files when you're done using them so that the system resources are freed.

Also, you might not find that all the information that you've written to the file has actually been written to disk until the file is closed. (You can force this with a call to `fflush()`.)

When your program exits normally, it closes all open files for you. Lots of times, though, you'll have a long-running program, and it'd be better to close the files before then. In any case, not closing a file you've opened makes you look bad. So, remember to `fclose()` your file when you're done with it!

Return Value

On success, `0` is returned. Typically no one checks for this. On error `EOF` is returned. Typically no one checks for this, either.

Example

```
FILE *fp;

fp = fopen("spoonDB.dat", r"); // (you should error-check this)
sort_spoon_database(fp);
fclose(fp);   // pretty simple, huh.
```

See Also

`fopen()`

`fflush()`

Process all buffered I/O for a stream right now

Synopsis

```
#include <stdio.h>

int fflush(FILE *stream);
```

Description

When you do standard I/O, as mentioned in the section on the `setvbuf()` function, it is usually stored in a buffer until a line has been entered or the buffer is full or the file is closed. Sometimes, though, you really want the output to happen *right this second*, and not wait around in the buffer. You can force this to happen by calling `fflush()`.

The advantage to buffering is that the OS doesn't need to hit the disk every time you call `fprintf()`. The disadvantage is that if you look at the file on the disk after the `fprintf()` call, it might not have actually been written to yet. ("I called `fputs()`, but the file is still zero bytes long! Why?!") In virtually all circumstances, the advantages of buffering outweigh the disadvantages; for those other circumstances, however, use `fflush()`.

Note that `fflush()` is only designed to work on output streams according to the spec. What will happen if you try it on an input stream? Use your spooky voice: *who knoooooows!*

Return Value

On success, `fflush()` returns zero. If there's an error, it returns `EOF` and sets the error condition for the stream (see `ferror()`.)

Example

In this example, we're going to use the carriage return, which is `'\r'`. This is like newline (`'\n'`), except that it doesn't move to the next line. It just returns to the front of the current line.

What we're going to do is a little text-based status bar like so many command line programs implement. It'll do a countdown from 10 to 0 printing over itself on the same line.

What is the catch and what does this have to do with `fflush()`? The catch is that the terminal is most likely "line buffered" (see the section on `setvbuf()` for more info), meaning that it won't actually display anything until it prints a newline. But we're not printing newlines; we're just printing carriage returns, so we need a way to force the output to occur even though we're on the same line. Yes, it's `fflush()`!

```
#include <stdio.h>
#include <unistd.h> // for prototype for sleep()

int main(void)
{
    int count;

    for(count = 10; count >= 0; count--) {
        printf("\rSeconds until launch: ");  // lead with a CR
        if (count > 0)
            printf("%2d", count);
        else
            printf("blastoff!\n");
```

```

        // force output now!!
        fflush(stdout);

        // the sleep() function is non-standard, but virtually every
        // system implements it--it simply delays for the specificed
        // number of seconds:
        sleep(1);
    }
}
```

See Also

`setbuf()`, `setvbuf()`

fopen()

Opens a file for reading or writing

Synopsis

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
```

Description

The `fopen()` opens a file for reading or writing.

Parameter `path` can be a relative or fully-qualified path and file name to the file in question.

Parameter `mode` tells `fopen()` how to open the file (reading, writing, or both), and whether or not it's a binary file. Possible modes are:

Mode	Description
`r`	Open the file for reading (read-only).
`w`	Open the file for writing (write-only). The file is created if it doesn't exist.
`r+`	Open the file for reading and writing. The file has to already exist.
`w+`	Open the file for writing and reading. The file is created if it doesn't already exist.
`a`	Open the file for append. This is just like opening a file for writing, but it positions the file pointer at the end of the file, so the next write appends to the end. The file is created if it doesn't exist.
`a+`	Open the file for reading and appending. The file is created if it doesn't exist.

Any of the modes can have the letter "`b`" appended to the end, as is "`wb`" ("write binary"), to signify that the file in question is a *binary* file. ("Binary" in this case generally means that the file contains non-alphanumeric characters that look like garbage to human eyes.) Many systems (like Unix) don't differentiate between binary and non-binary files, so the "`b`" is extraneous. But if your data is binary, it doesn't hurt to throw the "`b`" in there, and it might help someone who is trying to port your code to another system.

The macro `FOPEN_MAX` tells you how many streams (at least) you can have open at once.

The macro `FILENAME_MAX` tells you what the longest valid filename can be. Don't go crazy, now.

Return Value

`fopen()` returns a `FILE*` that can be used in subsequent file-related calls.

If something goes wrong (e.g. you tried to open a file for read that didn't exist), `fopen()` will return `NULL`.

Example

```
int main(void)
{
    FILE *fp;
```

```c

    if ((fp = fopen("datafile.dat", "r")) == NULL) {
        printf("Couldn't open datafile.dat for reading\n");
        exit(1);
    }

    // fp is now initialized and can be read from it
}
```

See Also

`fclose()`, `freopen()`

freopen()

Reopen an existing `FILE*`, associating it with a new path

Synopsis

```
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Description

Let's say you have an existing `FILE*` stream that's already open, but you want it to suddenly use a different file than the one it's using. You can use `freopen()` to "re-open" the stream with a new file.

Why on Earth would you ever want to do that? Well, the most common reason would be if you had a program that normally would read from `stdin`, but instead you wanted it to read from a file. Instead of changing all your `scanf()`s to `fscanf()`s, you could simply reopen `stdin` on the file you wanted to read from.

Another usage that is allowed on some systems is that you can pass `NULL` for `filename`, and specify a new `mode` for `stream`. So you could change a file from "`r+`" (read and write) to just "`r`" (read), for instance. It's implementation dependent which modes can be changed.

When you call `freopen()`, the old `stream` is closed. Otherwise, the function behaves just like the standard `fopen()`.

Return Value

`freopen()` returns `stream` if all goes well.

If something goes wrong (e.g. you tried to open a file for read that didn't exist), `freopen()` will return `NULL`.

Example

```
#include <stdio.h>

int main(void)
{
    int i, i2;

    scanf("%d", &i); // read i from stdin

    // now change stdin to refer to a file instead of the keyboard
    freopen("someints.txt", "r", stdin);

    scanf("%d", &i2); // now this reads from the file "someints.txt"

    printf("Hello, world!\n"); // print to the screen

    // change stdout to go to a file instead of the terminal:
    freopen("output.txt", "w", stdout);

    printf("This goes to the file \"output.txt\"\n");

    // this is allowed on some systems--you can change the mode of a file:
```

```
    freopen(NULL, "wb", stdout); // change to "wb" instead of "w"
}
```

See Also

`fclose()`, `fopen()`

setbuf(), setvbuf()

Configure buffering for standard I/O operations

Synopsis

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Description

Now brace yourself because this might come as a bit of a surprise to you: when you `printf()` or `fprintf()` or use any I/O functions like that, *it does not normally work immediately*. For the sake of efficiency, and to irritate you, the I/O on a `FILE*` stream is buffered away safely until certain conditions are met, and only then is the actual I/O performed. The functions `setbuf()` and `setvbuf()` allow you to change those conditions and the buffering behavior.

So what are the different buffering behaviors? The biggest is called "full buffering", wherein all I/O is stored in a big buffer until it is full, and then it is dumped out to disk (or whatever the file is). The next biggest is called "line buffering"; with line buffering, I/O is stored up a line at a time (until a newline (`'\n'`) character is encountered) and then that line is processed. Finally, we have "unbuffered", which means I/O is processed immediately with every standard I/O call.

You might have seen and wondered why you could call `putchar()` time and time again and not see any output until you called `putchar('\n')`; that's right—`stdout` is line-buffered!

Since `setbuf()` is just a simplified version of `setvbuf()`, we'll talk about `setvbuf()` first.

The `stream` is the `FILE*` you wish to modify. The standard says you *must* make your call to `setvbuf()` *before* any I/O operation is performed on the stream, or else by then it might be too late.

The next argument, `buf` allows you to make your own buffer space (using `malloc()` or just a `char` array) to use for buffering. If you don't care to do this, just set `buf` to `NULL`.

Now we get to the real meat of the function: `mode` allows you to choose what kind of buffering you want to use on this `stream`. Set it to one of the following:

Mode	Description
`_IOFBF`	`stream` will be fully buffered.
`_IOLBF`	`stream` will be line buffered.
`_IONBF`	`stream` will be unbuffered.

Finally, the `size` argument is the size of the array you passed in for `buf`...unless you passed `NULL` for `buf`, in which case it will resize the existing buffer to the size you specify.

Now what about this lesser function `setbuf()`? It's just like calling `setvbuf()` with some specific parameters, except `setbuf()` doesn't return a value. The following example shows the equivalency:

```
// these are the same:
setbuf(stream, buf);
setvbuf(stream, buf, _IOFBF, BUFSIZ); // fully buffered

// and these are the same:
```

```
    setbuf(stream, NULL);
    setvbuf(stream, NULL, _IONBF, BUFSIZ); // unbuffered
```

Return Value

`setvbuf()` returns zero on success, and nonzero on failure. `setbuf()` has no return value.

Example

```
FILE *fp;
char lineBuf[1024];

fp = fopen("somefile.txt", "r");
setvbuf(fp, lineBuf, _IOLBF, 1024);  // set to line buffering
// ...
fclose(fp);

fp = fopen("another.dat", "rb");
setbuf(fp, NULL); // set to unbuffered
// ...
fclose(fp);
```

See Also

`fflush()`

`printf()`, `fprintf()`, `sprintf()`, `snprintf()`

Print a formatted string to the console or to a file.

Synopsis

```
#include <stdio.h>

int printf(const char *format, ...);

int fprintf(FILE *stream, const char *format, ...);

int sprintf(char * restrict s, const char * restrict format, ...);

int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
```

Description

These functions print formatted output to a variety of destinations.

Function	Output Destination
`printf()`	Print to console (screen by default, typically).
`fprintf()`	Print to a file.
`sprintf()`	Print to a string.
`snprintf()`	Print to a string (safely).

The only differences between these is are the leading parameters that you pass to them before the `format` string.

Function	What you pass before `format`
`printf()`	Nothing comes before `format`.
`fprintf()`	Pass a `FILE*`.
`sprintf()`	Pass a `char*` to a buffer to print into.
`snprintf()`	Pass a `char*` to the buffer and a maximum buffer length.

The `printf()` function is legendary as being one of the most flexible outputting systems ever devised. It can also get a bit freaky here or there, most notably in the `format` string. We'll take it a step at a time here.

The easiest way to look at the format string is that it will print everything in the string as-is, *unless* a character has a percent sign (`%`) in front of it. That's when the magic happens: the next argument in the `printf()` argument list is printed in the way described by the percent code. These percent codes are called *format specifiers*.

Here are the most common format specifiers.

Specifier	Description
`%d`	Print the next argument as a signed decimal number, like `3490`. The argument printed this way should be an `int`, or something that gets promoted to `int`.
`%f`	Print the next argument as a signed floating point number, like `3.14159`. The argument printed this way should be a `double`, or something that gets promoted to a `double`.
`%c`	Print the next argument as a character, like `'B'`. The argument printed this way should be a `char` variant.

Specifier	Description
`%s`	Print the next argument as a string, like `"Did you remember your mittens?"`. The argument printed this way should be a `char*` or `char[]`.
`%%`	No arguments are converted, and a plain old run-of-the-mill percent sign is printed. This is how you print a '%' using `printf()`.

So those are the basics. I'll give you some more of the format specifiers in a bit, but let's get some more breadth before then. There's actually a lot more that you can specify in there after the percent sign.

For one thing, you can put a field width in there—this is a number that tells `printf()` how many spaces to put on one side or the other of the value you're printing. That helps you line things up in nice columns. If the number is negative, the result becomes left-justified instead of right-justified. Example:

```
printf("%10d", x);  /* prints X on the right side of the 10-space field */
printf("%-10d", x); /* prints X on the left side of the 10-space field */
```

If you don't know the field width in advance, you can use a little kung-foo to get it from the argument list just before the argument itself. Do this by placing your seat and tray tables in the fully upright position. The seatbelt is fastened by placing the—cough. I seem to have been doing way too much flying lately. Ignoring that useless fact completely, you can specify a dynamic field width by putting a `*` in for the width. If you are not willing or able to perform this task, please notify a flight attendant and we will reseat you.

```
int width = 12;
int value = 3490;

printf("%*d\n", width, value);
```

You can also put a "0" in front of the number if you want it to be padded with zeros:

```
int x = 17;
printf("%05d", x);  /* "00017" */
```

When it comes to floating point, you can also specify how many decimal places to print by making a field width of the form "`x.y`" where `x` is the field width (you can leave this off if you want it to be just wide enough) and `y` is the number of digits past the decimal point to print:

```
float f = 3.1415926535;

printf("%.2f", f);  /* "3.14" */
printf("%7.3f", f); /* "  3.141" <-- 7 spaces across */
```

Ok, those above are definitely the most common uses of `printf()`, but let's get *total coverage*.

Format Specifier Layout

Technically, the layout of the format specifier is these things in this order:

1. `%`, followed by…
2. Optional: zero or more flags, left justify, leading zeros, etc.
3. Optional: Field width, how wide the output field should be.
4. Optional: Precision, or how many decimal places to print.
5. Optional: Length modifier, for printing things bigger than `int` or `double`.
6. Conversion specifier, like `d`, `f`, etc.

In short, the whole format specifier is laid out like this:

```
%[flags][fieldwidth][.precision][lengthmodifier]conversionspecifier
```

What could be easier?

Conversion Specifiers

Let's talk conversion specifiers first. Each of the following specifies what type it can print, but it can also print anything that gets promoted to that type. For example, `%d` can print `int`, `short`, and `char`.

Conversion Specifier	Description
`d`	Print an `int` argument as a decimal number.
`i`	Identical to `d`.
`o`	Print an `unsigned int` in octal (base 8).
`u`	Print an `unsigned int` in decimal.
`x`	Print an `unsigned int` in hexadecimal with lowercase letters.
`X`	Print an `unsigned int` in hexadecimal with uppercase letters.
`f`	Print a `double` in decimal notation. Infinity is printed as `infinity` or `inf`, and NaN is printed as `nan`, any of which could have a leading minus sign.
`F`	Same as `f`, except it prints out `INFINITY`, `INF`, or `NAN` in all caps.
`e`	Print a number in scientific notation, e.g. `1.234e56`. Does infinity and NaN like `f`.
`E`	Just like `e`, except prints the exponent `E` (and infinity and NaN) in uppercase.
`g`	Print small numbers like `f` and large numbers like `e`. See note below.
`G`	Print small numbers like `F` and large numbers like `E`. See note below.
`a`	Print a `double` in hexadecimal form `0xh.hhhhpd` where `h` is a lowercase hex digit and `d` is a decimal exponent of 2. Infinity and NaN in the form of `f`. More below.
`A`	Like `a` except everything's uppercase.
`c`	Convert `int` argument to `unsigned char` and print as a character.
`s`	Print a string starting at the given `char*`.
`p`	Print a `void*` out as a number, probably the numeric address, possibly in hex.
`n`	Store the number of characters written so far in the given `int*`. Doesn't print anything. See below.
`%`	Print a literal percent sign.

Note on `%a` and `%A` When printing floating point numbers in hex form, there is one number before the decimal point, and the rest of are out to the precision.

```
double pi = 3.14159265358979;

printf("%.3a\n", pi);  // 0x1.922p+1
```

C can choose the leading number in such a way to ensure subsequent digits align to 4-bit boundaries.

If the precision is left out and the macro `FLT_RADIX` is a power of 2, enough precision is used to represent the number exactly. If `FLT_RADIX` is not a power of two, enough precision is used to be able to tell any two floating values apart.

If the precision is `0` and the `#` flag isn't specified, the decimal point is omitted.

Note on `%g` and `%G` The gist of this is to use scientific notation when the number gets too "extreme", and regular decimal notation otherwise.

The exact behavior for whether these print as `%f` or `%e` depends on a number of factors:

If the number's exponent is greater than or equal to -4 **and** the precision is greater than the exponent, we use `%f`. In this case, the precision is converted according to $p = p - (x + 1)$, where p is the specified precision and x is the exponent.

Otherwise we use `%e`, and the precision becomes $p - 1$.

Trailing zeros in the decimal portion are removed. And if there are none left, the decimal point is removed, too. All this unless the # flag is specified.

Note on `%n` This specifier is cool and different, and rarely needed. It doesn't actually print anything, but stores the number of characters printed so far in the next pointer argument in the list.

```
int numChars;
float a = 3.14159;
int b = 3490;

printf("%f %d%n\n", a, b, &numChars);
printf("The above line contains %d characters.\n", numChars);
```

The above example will print out the values of `a` and `b`, and then store the number of characters printed so far into the variable `numChars`. The next call to `printf()` prints out that result.

```
3.141590 3490
The above line contains 13 characters
```

Length Modifiers

You can stick a *length* modifier in front of each of the conversion specifiers, if you want. most of those format specifiers work on `int` or `double` types, but what if you want larger or smaller types? That's what these are good for.

For example, you could print out a long long int with the `ll` modifier:

```
long long int x = 3490;

printf("%lld\n", x);  // 3490
```

Length Modifier	Conversion Specifier	Description
`hh`	`d`, `i`, `o`, `u`, `x`, `X`	Convert argument to `char` (signed or unsigned as appropriate) before printing.
`h`	`d`, `i`, `o`, `u`, `x`, `X`	Convert argument to `short int` (signed or unsigned as appropriate) before printing.
`l`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `long int` (signed or unsigned as appropriate).
`ll`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `long long int` (signed or unsigned as appropriate).
`j`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `intmax_t` or `uintmax_t` (as appropriate).
`z`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `size_t`.
`t`	`d`, `i`, `o`, `u`, `x`, `X`	Argument is a `ptrdiff_t`.
`L`	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	Argument is a `long double`.
`l`	`c`	Argument is in a `wint_t`, a wide character.
`l`	`s`	Argument is in a `wchar_t*`, a wide character string.
`hh`	`n`	Store result in `signed char*` argument.
`h`	`n`	Store result in `short int*` argument.
`l`	`n`	Store result in `long int*` argument.
`ll`	`n`	Store result in `long long int*` argument.
`j`	`n`	Store result in `intmax_t*` argument.
`z`	`n`	Store result in `size_t*` argument.
`t`	`n`	Store result in `ptrdiff_t*` argument.

Precision

In front of the length modifier, you can put a precision, which generally means how many decimal places you want on your floating point numbers.

To do this, you put a decimal point (`.`) and the decimal places afterward.

For example, we could print π rounded to two decimal places like this:

```
double pi = 3.14159265358979;

printf("%.2f\n", pi);  // 3.14
```

Conversion Specifier	Precision Value Meaning
`d`, `i`, `o`, `u`, `x`, `X`	For integer types, minimum number of digits (will pad with leading zeros)
`a`, `e`, `f`, `A`, `E`, `F`	For floating types, the precision is the number of digits past the decimal.
`g`, `G`	For floating types, the precision is the number of significant digits printed.
`s`	The maximum number of bytes (not multibyte characters!) to be written.

If no number is specified in the precision after the decimal point, the precision is zero.

If an `*` is specified after the decimal, something amazing happens! It means the `int` argument to `printf()` before the number to be printed holds the precision. You can use this if you don't know the precision at compile time.

```
int precision;
double pi = 3.14159265358979;

printf("Enter precision: "); fflush(stdout);
scanf("%d", &precision);

printf("%.*f\n", precision, pi);
```

Which gives:

```
Enter precision: 4
3.1416
```

Field Width

In front of the optional precision, you can indicate a field width. This is a decimal number that indicates how wide the region should be in which the argument is printed. The region is padding with leading (or trailing) spaces to make sure it's wide enough.

If the field width specified is too small to hold the output, it is ignored.

As a preview, you can give a negative field width to justify the item the other direction.

So let's print a number in a field of width 10. We'll put some angle brackets around it so we can see the padding spaces in the output.

```
printf("<<%10d>>\n", 3490);   // right justified
printf("<<%-10d>>\n", 3490);  // left justified
```

```
<<      3490>>
<<3490      >>
```

Like with the precision, you can use an asterisk (`*`) as the field width

```
int field_width;
int val = 3490;

printf("Enter field_width: "); fflush(stdout);
scanf("%d", &field_width);

printf("<<%*d>>\n", field_width, val);
```

Flags

Before the field width, you can put some optional flags that further control the output of the subsequent fields. We just saw that the - flag can be used to left- or right-justify fields. But there are plenty more!

Flag	Description
-	For a field width, left justify in the field (right is default).
+	If the number is signed, always prefix a + or - on the front.
	(Space.) If the number is signed, prefix a space for positive, or a - for negative.
0	Pad the right-justified field with leading zeros instead of leading spaces.
#	Print using an alternate form. See below.

For example, we could pad a hexadecimal number with leading zeros to a field width of 8 with:

```
printf("%08x\n", 0x1234);  // 00001234
```

The # "alternate form" result depends on the conversion specifier.

Conversion Specifier	Alternate Form (#) Meaning
o	Increase precision of a non-zero number just enough to get one leading 0 on the octal number.
x	Prefix a non-zero number with 0x.
X	Same as x, except capital 0X.
a, e, f	Always print a decimal point, even if nothing follows it.
A, E, F	Identical to a, e, f.
g, G	Always print a decimal point, even if nothing follows it, and keep trailing zeros.

sprintf() and snprintf() Details

Both sprintf() and snprintf() have the quality that if you pass in NULL as the buffer, nothing is written—but you can still check the return value to see how many characters *would* have been written.

snprintf() **always** terminates the string with a NUL character. So if you try to write out more than the maximum specified characters, the universe ends.

Just kidding. If you do, snprintf() will write $n - 1$ characters so that it has enough room to write the terminator at the end.

Return Value

Returns the number of characters outputted, or a negative number on error.

Example

```
int a = 100;
float b = 2.717;
```

```
char *c = "beej!";
char d = 'X';
int e = 5;

printf("%d", a); /* "100"      */
printf("%f", b); /* "2.717000" */
printf("%s", c); /* "beej!"    */
printf("%c", d); /* "X"        */
printf("110%%"); /* "110%"     */

printf("%10d\n", a);   /* "       100" */
printf("%-10d\n", a);  /* "100       " */
printf("%*d\n", e, a); /* "  100"      */
printf("%.2f\n", b);   /* "2.71"       */

printf("%hhd\n", c); /* "88" <-- ASCII code for 'X' */

printf("%5d %5.2f %c\n", a, b, d); /* "  100  2.71 X" */
```

See Also

`sprintf()`, `vprintf()`

scanf(), fscanf(), sscanf()

Read formatted string, character, or numeric data from the console or from a file.

Synopsis

```
#include <stdio.h>

int scanf(const char *format, ...);

int fscanf(FILE *stream, const char *format, ...);

int sscanf(const char * restrict s, const char * restrict format, ...);
```

Description

These functions read formatted output from a variety of sources.

Function	Input Source
`scanf()`	Read from the console (keyboard by default, typically).
`fscanf()`	Read from a file.
`sscanf()`	Read from a string.

The only differences between these is are the leading parameters that you pass to them before the `format` string.

Function	What you pass before `format`
`scanf()`	Nothing comes before `format`.
`fscanf()`	Pass a `FILE*`.
`sscanf()`	Pass a `char*` to a buffer to read from.

The `scanf()` family of functions reads data from the console or from a `FILE` stream, parses it, and stores the results away in variables you provide in the argument list.

The format string is very similar to that in `printf()` in that you can tell it to read a `"%d"`, for instance for an `int`. But it also has additional capabilities, most notably that it can eat up other characters in the input that you specify in the format string.

But let's start simple, and look at the most basic usage first before plunging into the depths of the function. We'll start by reading an `int` from the keyboard:

```
int a;

scanf("%d", &a);
```

`scanf()` obviously needs a pointer to the variable if it is going to change the variable itself, so we use the address-of operator to get the pointer.

In this case, `scanf()` walks down the format string, finds a "`%d`", and then knows it needs to read an integer and store it in the next variable in the argument list, `a`.

Here are some of the other format specifiers you can put in the format string:

Format Specifier	Description
`%d`	Reads an integer to be stored in an `int`. This integer can be signed.
`%u`	Reads an integer to be stored in an `unsigned int`.
`%f`	Reads a floating point number, to be stored in a `float`.
`%s`	Reads a string up to the first whitespace character.
`%c`	Reads a `char`.

And that's the end of the story!

Ha! Just kidding. If you've just arrived from the `printf()` page, you know there's a near-infinite amount of additional material.

Consuming Other Characters

`scanf()` will move along the format string matching any characters you include.

For example, you could read a hyphenated date like so:

```
scanf("%u-%u-%u", &yyyy, &mm, &dd);
```

In that case, `scanf()` will attempt to consume an unsigned decimal number, then a hyphen, then another unsigned number, then another hypen, then another unsigned number.

If it fails to match at any point (e.g. the user entered "foo"), `scanf()` will bail without consuming the offending characters.

And it will return the number of variables successfully converted. In the example above, if the user entered a valid string, `scanf()` would return `3`, one for each variable successfully read.

Problems with `scanf()`

I (and the C FAQ and a lot of people) recommend *against* using `scanf()` to read directly from the keyboard. It's too easy for it to stop consuming characters when the user enters some bad data.

If you have data in a file and you're confident it's in good shape, `fscanf()` can be really useful.

But in the case of the keyboard or file, you can always use `fgets()` to read a complete line into a buffer, and then use `sscanf()` to scan things out of the buffer. This gives you the best of both worlds.

The Deep Details

Let's check out what a `scanf()`

And here are some more codes, except these don't tend to be used as often. You, of course, may use them as often as you wish!

First, the format string. Like we mentioned, it can hold ordinary characters as well as `%` format specifiers. And whitespace characters.

Whitespace characters have a special role: a whitespace character will cause `scanf()` to consume as many whitespace characters as it can up to the next non-whitespace character. You can use this to ignore all leading or trailing whitespace.

Also, all format specifiers except for `s`, `c`, and `[` automatically consume leading whitespace.

But I know what you're thinking: the meat of this function is in the format specifiers. What do those look like?

These consist of the following, in sequence:

1. A % sign
2. Optional: an * to suppress assignment—more later
3. Optional: a field width—max characters to read
4. Optional: length modifier, for specifying longer or shorter types
5. A conversion specifier, like d or f indicating the type to read

The Conversion Specifier

Let's start with the best and last: the *conversion specifier*.

This is the part of the format specifier that tells us what type of variable scanf() should be reading into, like %d or %f.

Conversion Specifier	Description
d	Matches a decimal int. Can have a leading sign.
i	Like d, except will handle it if you put a leading 0x (hex) or 0 (octal) on the number.
o	Matches an octal (base 8) unsigned int. Leading zeros are ignored.
u	Matches a decimal unsigned int.
x	Matches a hex (base 16) unsigned int.
f	Match a floating point number (or scientific notation, or anything strtod() can handle).
c	Match a char, or mutiple chars if a field width is given.
s	Match a sequence of non-whitespace chars.
[	Match a sequence of characters from a set. The set ends with]. More below.
p	Match a pointer, the opposite of %p for printf().
n	Store the number of characters written so far in the given int*. Doesn't consume anything.
%	Match a literal percent sign.

All of the following are equivalent to the f specifier: a, e, g, A, E, F, G.

And capital X is equivalent to lowercase x.

The Scanset %[] Conversion Specifier This is about the weirdest format specifier there is. It allows you to specify a set of characters (the *scanset*) to be stored away (likely in an array of chars). Conversion stops when a character that is not in the set is matched.

For example, %[0-9] means "match all numbers zero through nine." And %[AD-G34] means "match A, D through G, 3, or 4".

Now, to convolute matters, you can tell scanf() to match characters that are *not* in the set by putting a caret (^) directly after the %[and following it with the set, like this: %[^A-C], which means "match all characters that are *not* A through C."

To match a close square bracket, make it the first character in the set, like this: %[]A-C] or %[^]A-C]. (I added the "A-C" just so it was clear that the "]" was first in the set.)

To match a hyphen, make it the last character in the set, e.g. to match A-through-C or hyphen: %[A-C-].

So if we wanted to match all letters *except* "%", "^", "]", "B", "C", "D", "E", and "-", we could use this format string: %[^]%^B-E-].

Got it? Now we can go onto the next func—no wait! There's more! Yes, still more to know about scanf(). Does it never end? Try to imagine how I feel writing about it!

The Length Modifier

So you know that "%d" stores into an int. But how do you store into a long, short, or double?

Well, like in `printf()`, you can add a modifier before the type specifier to tell `scanf()` that you have a longer or shorter type. The following is a table of the possible modifiers:

Length Modifier	Conversion Specifier	Description
`hh`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `char` (signed or unsigned as appropriate) before printing.
`h`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `short int` (signed or unsigned as appropriate) before printing.
`l`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `long int` (signed or unsigned as appropriate).
`ll`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `long long int` (signed or unsigned as appropriate).
`j`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `intmax_t` or `uintmax_t` (as appropriate).
`z`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `size_t`.
`t`	`d`, `i`, `o`, `u`, `x`, `X`	Convert input to `ptrdiff_t`.
`L`	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	Convert input to `long double`.
`l`	`c`,`s`,`[`	Convert input to `wchar_t`, a wide character.
`l`	`s`	Argument is in a `wchar_t*`, a wide character string.
`hh`	`n`	Store result in `signed char*` argument.
`h`	`n`	Store result in `short int*` argument.
`l`	`n`	Store result in `long int*` argument.
`ll`	`n`	Store result in `long long int*` argument.
`j`	`n`	Store result in `intmax_t*` argument.
`z`	`n`	Store result in `size_t*` argument.
`t`	`n`	Store result in `ptrdiff_t*` argument.

Field Widths

The field width generally allows you to specify a maximum number of characters to consume. If the thing you're trying to match is shorter than the field width, that input will stop being processed before the field width is reached.

So a string will stop being consumed when whitespace is found, even if fewer than the field width characters are matched.

And a float will stop being consumed at the end of the number, even if fewer characters than the field width are matched.

But `%c` is an interesting one—it doesn't stop consuming characters on anything. So it'll go exactly to the field width. (Or 1 character if no field width is given.)

Skip Input with `*`

If you put an `*` in the format specifier, it tells `scanf()` do to the conversion specified, but not store it anywhere. It simply discards the data as it reads it. This is what you use if you want `scanf()` to eat some data but you don't want to store it anywhere; you don't give `scanf()` an argument for this conversion.

```
// Read 3 ints, but discard the middle one
scanf("%d %*d %d", &int1, &int3);
```

Return Value

`scanf()` returns the number of items assigned into variables. Since assignment into variables stops when given invalid input for a certain format specifier, this can tell you if you've input all your data correctly.

Also, `scanf()` returns `EOF` on end-of-file.

Example

```c
int a;
long int b;
unsigned int c;
float d;
double e;
long double f;
char s[100];

scanf("%d", &a);  // store an int
scanf(" %d", &a); // eat any whitespace, then store an int
scanf("%s", s); // store a string
scanf("%Lf", &f); // store a long double

// store an unsigned, read all whitespace, then store a long int:
scanf("%u %ld", &c, &b);

// store an int, read whitespace, read "blendo", read whitespace,
// and store a float:
scanf("%d blendo %f", &a, &d);

// read all whitespace, then store all characters up to a newline
scanf(" %[^\n]", s);

// store a float, read (and ignore) an int, then store a double:
scanf("%f %*d %lf", &d, &e);

// store 10 characters:
scanf("%10c", s);
```

See Also

`sscanf()`, `vscanf()`, `vsscanf()`, `vfscanf()`

vprintf(), vfprintf(), vsprintf(), vsnprintf()

printf() variants using variable argument lists (va_list)

Synopsis

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

int vprintf(const char * restrict format, va_list arg);

int vfprintf(FILE * restrict stream, const char * restrict format,
             va_list arg);

int vsprintf(char * restrict s, const char * restrict format, va_list arg);

int vsnprintf(char * restrict s, size_t n, const char * restrict format,
              va_list arg);
```

Description

These are just like the printf() variants except instead of taking an actual variable number of arguments, they take a fixed number—the last of which is a va_list that refers to the variable arguments.

Like with printf(), the different variants send output different places.

Function	Output Destination
vprintf()	Print to console (screen by default, typically).
vfprintf()	Print to a file.
vsprintf()	Print to a string.
vsnprintf()	Print to a string (safely).

Both vsprintf() and vsnprintf() have the quality that if you pass in NULL as the buffer, nothing is written—but you can still check the return value to see how many characters *would* have been written.

vsnprintf() **always** terminates the string with a NUL character. So if you try to write out more than the maximum specified characters, a massive interstellar war breaks out.

Just kidding. If you do, vsnprintf() will write $n - 1$ characters so that it has enough room to write the terminator at the end.

As for why in the heck would you ever want to do this, the most common reason is to create your own specialized versions of printf()-type functions, piggybacking on all that printf() functionality goodness.

See the example for an example, predictably.

Return Value

vprintf() and vfprintf() return the number of characters printed, or a negative value on error.

vsprintf() returns the number of characters printed to the buffer, not counting the NUL terminator, or a negative value if an error occurred.

vnsprintf() returns the number of characters printed to the buffer. Or the number that *would* have been printed if the buffer had been large enough.

Example

In this example, we make our own version of `printf()` called `logger()` that timestamps output. Notice how the calls to `logger()` have all the bells and whistles of `printf()`.

```c
#include <stdio.h>
#include <stdarg.h>
#include <time.h>

int logger(char *format, ...)
{
    va_list va;
    time_t now_secs = time(NULL);
    struct tm *now = gmtime(&now_secs);

    // Output timestamp in format "YYYY-MM-DD hh:mm:ss : "
    printf("%04d-%02d-%02d %02d:%02d:%02d : ",
        now->tm_year + 1900, now->tm_mon + 1, now->tm_mday,
        now->tm_hour, now->tm_min, now->tm_sec);

    va_start(va, format);
    int result = vprintf(format, va);
    va_end(va);

    printf("\n");

    return result;
}

int main(void)
{
    int x = 12;
    float y = 3.2;

    logger("Hello!");
    logger("x = %d and y = %.2f", x, y);
}
```

See Also

`printf()`

`vscanf()`, `vfscanf()`, `vsscanf()`

`scanf()` variants using variable argument lists (`va_list`)

Synopsis

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

int vscanf(const char * restrict format, va_list arg);

int vfscanf(FILE * restrict stream, const char * restrict format,
            va_list arg);

int vsscanf(const char * restrict s, const char * restrict format,
            va_list arg);
```

Description

These are just like the `scanf()` variants except instead of taking an actual variable number of arguments, they take a fixed number—the last of which is a `va_list` that refers to the variable arguments.

Function	Input Source
`vscanf()`	Read from the console (keyboard by default, typically).
`vfscanf()`	Read from a file.
`vsscanf()`	Read from a string.

Like with the `vprintf()` functions, this would be a good way to add additional functionality that took advantage of the power `scanf()` has to offer.

Return Value

Returns the number of items successfully scanned, or `EOF` on end-of-file or error.

Example

I have to admit I was wracking my brain to think of when you'd ever want to use this. The best example I could find was one on Stack Overflow[159] that error-checks the return value from `scanf()` against the expected. A variant of that is shown below.

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

int error_check_scanf(int expected_count, char *format, ...)
{
    va_list va;

    va_start(va, format);
    int count = vscanf(format, va);
    va_end(va);

```

[159] https://stackoverflow.com/questions/17017331/c99-vscanf-for-dummies/17018046#17018046

```
    // This line will crash the program if the condition is false:
    assert(count == expected_count);

    return count;
}

int main(void)
{
    int a, b;
    float c;

    error_check_scanf(3, "%d, %d/%f", &a, &b, &c);
    error_check_scanf(2, "%d", &a);
}
```

See Also

`scanf()`

getc(), fgetc(), getchar()

Get a single character from the console or from a file.

Synopsis

```
#include <stdio.h>

int getc(FILE *stream);

int fgetc(FILE *stream);

int getchar(void);
```

Description

All of these functions in one way or another, read a single character from the console or from a `FILE`. The differences are fairly minor, and here are the descriptions:

`getc()` returns a character from the specified `FILE`. From a usage standpoint, it's equivalent to the same `fgetc()` call, and `fgetc()` is a little more common to see. Only the implementation of the two functions differs.

`fgetc()` returns a character from the specified `FILE`. From a usage standpoint, it's equivalent to the same `getc()` call, except that `fgetc()` is a little more common to see. Only the implementation of the two functions differs.

Yes, I cheated and used cut-n-paste to do that last paragraph.

`getchar()` returns a character from `stdin`. In fact, it's the same as calling `getc(stdin)`.

Return Value

All three functions return the `unsigned char` that they read, except it's cast to an `int`.

If end-of-file or an error is encountered, all three functions return `EOF`.

Example

```
// read all characters from a file, outputting only the letter 'b's
// it finds in the file

#include <stdio.h>

int main(void)
{
    FILE *fp;
    int c;

    fp = fopen("datafile.txt", "r"); // error check this!

    // this while-statement assigns into c, and then checks against EOF:

    while((c = fgetc(fp)) != EOF) {
        if (c == 'b') {
            putchar(c);
        }
```

```
    }

    fclose(fp);
}
```

See Also

gets(), fgets()

Read a string from console or file

Synopsis

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

Description

These are functions that will retrieve a newline-terminated string from the console or a file. In other normal words, it reads a line of text. The behavior is slightly different, and, as such, so is the usage. For instance, here is the usage of `gets()`:

Don't use `gets()`. In fact, as of C11, it ceases to exist! This is one of the rare cases of a function being *removed* from the standard.

Admittedly, rationale would be useful, yes? For one thing, `gets()` doesn't allow you to specify the length of the buffer to store the string in. This would allow people to keep entering data past the end of your buffer, and believe me, this would be Bad News.

I was going to add another reason, but that's basically the primary and only reason not to use `gets()`. As you might suspect, `fgets()` allows you to specify a maximum string length.

One difference here between the two functions: `gets()` will devour and throw away the newline at the end of the line, while `fgets()` will store it at the end of your string (space permitting).

Here's an example of using `fgets()` from the console, making it behave more like `gets()` (with the exception of the newline inclusion):

```
char s[100];
gets(s);  // don't use this--read a line (from stdin)
fgets(s, sizeof(s), stdin); // read a line from stdin
```

In this case, the `sizeof()` operator gives us the total size of the array in bytes, and since a `char` is a byte, it conveniently gives us the total size of the array.

Of course, like I keep saying, the string returned from `fgets()` probably has a newline at the end that you might not want. You can write a short function to chop the newline off—in fact, let's just roll that into our own version of `gets()`

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

char *safe_gets(char *s, int size)
{
    char *rv = fgets(s, size, stdin);

    if (rv == NULL)
        return NULL;

    int len = strlen(s);

    if (len > 0 && s[len-1] == '\n')  // if there's a newline
        s[len-1] = '\0';              // truncate the string
```

```
        return s;
    }
```

So, in summary, use `fgets()` to read a line of text from the keyboard or a file, and don't use `gets()`.

Return Value

Both `gets()` and `fgets()` return a pointer to the string passed.

On error or end-of-file, the functions return `NULL`.

Example

```
char s[100];

gets(s); // read from standard input (don't use this--use fgets()!)

fgets(s, sizeof(s), stdin); // read 100 bytes from standard input

fp = fopen("datafile.dat", "r"); // (you should error-check this)
fgets(s, 100, fp); // read 100 bytes from the file datafile.dat
fclose(fp);

fgets(s, 20, stdin); // read a maximum of 20 bytes from stdin
```

See Also

`getc()`, `fgetc()`, `getchar()`, `puts()`, `fputs()`, `ungetc()`

putc(), fputc(), putchar()

Write a single character to the console or to a file.

Synopsis

```
#include <stdio.h>

int putc(int c, FILE *stream);

int fputc(int c, FILE *stream);

int putchar(int c);
```

Description

All three functions output a single character, either to the console or to a `FILE`.

`putc()` takes a character argument, and outputs it to the specified `FILE`. `fputc()` does exactly the same thing, and differs from `putc()` in implementation only. Most people use `fputc()`.

`putchar()` writes the character to the console, and is the same as calling `putc(c, stdout)`.

Return Value

All three functions return the character written on success, or `EOF` on error.

Example

```
// print the alphabet

#include <stdio.h>

int main(void)
{
    char i;

    for(i = 'A'; i <= 'Z'; i++)
        putchar(i);

    putchar('\n'); // put a newline at the end to make it pretty
}
```

See Also

puts(), fputs()

Write a string to the console or to a file.

Synopsis

```
#include <stdio.h>

int puts(const char *s);

int fputs(const char *s, FILE *stream);
```

Description

Both these functions output a NUL-terminated string. `puts()` outputs to the console, while `fputs()` allows you to specify the file for output.

Return Value

Both functions return non-negative on success, or `EOF` on error.

Example

```
// read strings from the console and save them in a file

#include <stdio.h>

int main(void)
{
    FILE *fp;
    char s[100];

    fp = fopen("datafile.txt", "w"); // error check this!

    while(fgets(s, sizeof(s), stdin) != NULL) { // read a string
        fputs(s, fp);  // write it to the file we opened
    }

    fclose(fp);
}
```

See Also

ungetc()

Pushes a character back into the input stream.

Synopsis

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

Description

You know how `getc()` reads the next character from a file stream? Well, this is the opposite of that—it pushes a character back into the file stream so that it will show up again on the very next read from the stream, as if you'd never gotten it from `getc()` in the first place.

Why, in the name of all that is holy would you want to do that? Perhaps you have a stream of data that you're reading a character at a time, and you won't know to stop reading until you get a certain character, but you want to be able to read that character again later. You can read the character, see that it's what you're supposed to stop on, and then `ungetc()` it so it'll show up on the next read.

Yeah, that doesn't happen very often, but there we are.

Here's the catch: the standard only guarantees that you'll be able to push back *one character*. Some implementations might allow you to push back more, but there's really no way to tell and still be portable.

Return Value

On success, `ungetc()` returns the character you passed to it. On failure, it returns `EOF`.

Example

```
// read a piece of punctuation, then everything after it up to the next
// piece of punctuation. return the punctuation, and store the rest
// in a string
//
// sample input: !foo#bar*baz
// output:  return value: '!', s is "foo"
//          return value: '#', s is "bar"
//          return value: '*', s is "baz"
//

char read_punctstring(FILE *fp, char *s)
{
    char origpunct, c;

    origpunct = fgetc(fp);

    if (origpunct == EOF) // return EOF on end-of-file
        return EOF;

    while(c = fgetc(fp), !ispunct(c) && c != EOF) {
        *s++ = c; // save it in the string
    }
    *s = '\0'; // nul-terminate the string!

```

```
    // if we read punctuation last, ungetc it so we can fgetc it next
    // time:
    if (ispunct(c))
        ungetc(c, fp);
    }

    return origpunct;
}
```

See Also

`fgetc()`

`fread()`

Read binary data from a file.

Synopsis

```
#include <stdio.h>

size_t fread(void *p, size_t size, size_t nmemb, FILE *stream);
```

Description

You might remember that you can call `fopen()` with the "`b`" flag in the open mode string to open the file in "binary" mode. Files open in not-binary (ASCII or text mode) can be read using standard character-oriented calls like `fgetc()` or `fgets()`. Files open in binary mode are typically read using the `fread()` function.

All this function does is says, "Hey, read this many things where each thing is a certain number of bytes, and store the whole mess of them in memory starting at this pointer."

This can be very useful, believe me, when you want to do something like store 20 `int`s in a file.

But wait—can't you use `fprintf()` with the "`%d`" format specifier to save the `int`s to a text file and store them that way? Yes, sure. That has the advantage that a human can open the file and read the numbers. It has the disadvantage that it's slower to convert the numbers from `int`s to text and that the numbers are likely to take more space in the file. (Remember, an `int` is likely 4 bytes, but the string "12345678" is 8 bytes.)

So storing the binary data can certainly be more compact and faster to read.

Return Value

This function returns the number of items successfully read. If all requested items are read, the return value will be equal to that of the parameter `nmemb`. If EOF occurs, the return value will be zero.

To make you confused, it will also return zero if there's an error. You can use the functions `feof()` or `ferror()` to tell which one really happened.

Example

```
// read 10 numbers from a file and store them in an array

int main(void)
{
    int i;
    int n[10]
    FILE *fp;

    fp = fopen("binarynumbers.dat", "rb");
    fread(n, sizeof(int), 10, fp);  // read 10 ints
    fclose(fp);

    // print them out:
    for(i = 0; i < 10; i++)
        printf("n[%d] == %d\n", i, n[i]);
}
```

See Also

`fopen()`, `fwrite()`, `feof()`, `ferror()`

fwrite()

Write binary data to a file.

Synopsis

```
#include <stdio.h>

size_t fwrite(const void *p, size_t size, size_t nmemb, FILE *stream);
```

Description

This is the counterpart to the `fread()` function. It writes blocks of binary data to disk. For a description of what this means, see the entry for `fread()`.

Return Value

`fwrite()` returns the number of items successfully written, which should hopefully be `nmemb` that you passed in. It'll return zero on error.

Example

```
// save 10 random numbers to a file

int main(void)
{
    int i;
    int r[10];
    FILE *fp;

    // populate the array with random numbers:
    for(i = 0; i < 10; i++) {
        r[i] = rand();
    }

    // save the random numbers (10 ints) to the file
    fp = fopen("binaryfile.dat", "wb");
    fwrite(r, sizeof(int), 10, fp); // write 10 ints
    fclose(fp);
}
```

See Also

`fopen()`, `fread()`

`fgetpos()`, `fsetpos()`

Get the current position in a file, or set the current position in a file. Just like `ftell()` and `fseek()` for most systems.

Synopsis

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);

int fsetpos(FILE *stream, fpos_t *pos);
```

Description

These functions are just like `ftell()` and `fseek()`, except instead of counting in bytes, they use an *opaque* data structure to hold positional information about the file. (Opaque, in this case, means you're not supposed to know what the data type is made up of.)

On virtually every system (and certainly every system that I know of), people don't use these functions, using `ftell()` and `fseek()` instead. These functions exist just in case your system can't remember file positions as a simple byte offset.

Since the `pos` variable is opaque, you have to assign to it using the `fgetpos()` call itself. Then you save the value for later and use it to reset the position using `fsetpos()`.

Return Value

Both functions return zero on success, and `-1` on error.

Example

```
char s[100];
fpos_t pos;

fgets(s, sizeof(s), fp); // read a line from the file

fgetpos(fp, &pos);   // save the position

fgets(s, sizeof(s), fp); // read another line from the file

fsetpos(fp, &pos);   // now restore the position to where we saved
```

See Also

`fseek()`, `ftell()`, `rewind()`

fseek(), rewind()

Position the file pointer in anticipition of the next read or write.

Synopsis

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);

void rewind(FILE *stream);
```

Description

When doing reads and writes to a file, the OS keeps track of where you are in the file using a counter generically known as the file pointer. You can reposition the file pointer to a different point in the file using the `fseek()` call. Think of it as a way to randomly access you file.

The first argument is the file in question, obviously. `offset` argument is the position that you want to seek to, and `whence` is what that offset is relative to.

Of course, you probably like to think of the offset as being from the beginning of the file. I mean, "Seek to position 3490, that should be 3490 bytes from the beginning of the file." Well, it *can* be, but it doesn't have to be. Imagine the power you're wielding here. Try to command your enthusiasm.

You can set the value of `whence` to one of three things:

`whence`	Description
`SEEK_SET`	`offset` is relative to the beginning of the file. This is probably what you had in mind anyway, and is the most commonly used value for `whence`.
`SEEK_CUR`	`offset` is relative to the current file pointer position. So, in effect, you can say, "Move to my current position plus 30 bytes," or, "move to my current position minus 20 bytes."
`SEEK_END`	`offset` is relative to the end of the file. Just like `SEEK_SET` except from the other end of the file. Be sure to use negative values for `offset` if you want to back up from the end of the file, instead of going past the end into oblivion.

Speaking of seeking off the end of the file, can you do it? Sure thing. In fact, you can seek way off the end and then write a character; the file will be expanded to a size big enough to hold a bunch of zeros way out to that character.

Now that the complicated function is out of the way, what's this `rewind()` that I briefly mentioned? It repositions the file pointer at the beginning of the file:

```
fseek(fp, 0, SEEK_SET); // same as rewind()
rewind(fp);             // same as fseek(fp, 0, SEEK_SET)
```

Return Value

For `fseek()`, on success zero is returned; `-1` is returned on failure.

The call to `rewind()` never fails.

Example

```
fseek(fp, 100, SEEK_SET); // seek to the 100th byte of the file
fseek(fp, -30, SEEK_CUR); // seek backward 30 bytes from the current pos
```

```
fseek(fp, -10, SEEK_END); // seek to the 10th byte before the end of file

fseek(fp, 0, SEEK_SET);   // seek to the beginning of the file
rewind(fp);               // seek to the beginning of the file
```

See Also

`ftell()`, `fgetpos()`, `fsetpos()`

ftell()

Tells you where a particular file is about to read from or write to.

Synopsis

```
#include <stdio.h>

long ftell(FILE *stream);
```

Description

This function is the opposite of `fseek()`. It tells you where in the file the next file operation will occur relative to the beginning of the file.

It's useful if you want to remember where you are in the file, `fseek()` somewhere else, and then come back later. You can take the return value from `ftell()` and feed it back into `fseek()` (with `whence` parameter set to `SEEK_SET`) when you want to return to your previous position.

Return Value

Returns the current offset in the file, or `-1` on error.

Example

```
long pos;

// store the current position in variable "pos":
pos = ftell(fp);

// seek ahead 10 bytes:
fseek(fp, 10, SEEK_CUR);

// do some mysterious writes to the file
do_mysterious_writes_to_file(fp);

// and return to the starting position, stored in "pos":
fseek(fp, pos, SEEK_SET);
```

See Also

`fseek()`, `rewind()`, `fgetpos()`, `fsetpos()`

`feof()`, `ferror()`, `clearerr()`

Determine if a file has reached end-of-file or if an error has occurred.

Synopsis

```
#include <stdio.h>

int feof(FILE *stream);

int ferror(FILE *stream);

void clearerr(FILE *stream);
```

Description

Each `FILE*` that you use to read and write data from and to a file contains flags that the system sets when certain events occur. If you get an error, it sets the error flag; if you reach the end of the file during a read, it sets the EOF flag. Pretty simple really.

The functions `feof()` and `ferror()` give you a simple way to test these flags: they'll return non-zero (true) if they're set.

Once the flags are set for a particular stream, they stay that way until you call `clearerr()` to clear them.

Return Value

`feof()` and `ferror()` return non-zero (true) if the file has reached EOF or there has been an error, respectively.

Example

```
// read binary data, checking for eof or error
int main(void)
{
    int a;
    FILE *fp;

    fp = fopen("binaryints.dat", "rb");

    // read single ints at a time, stopping on EOF or error:

    while(fread(&a, sizeof(int), 1, fp), !feof(fp) && !ferror(fp)) {
        printf("I read %d\n", a);
    }

    if (feof(fp))
        printf("End of file was reached.\n");

    if (ferror(fp))
        printf("An error occurred.\n");

    fclose(fp);
}
```

See Also

`fopen()`, `fread()`

perror()

Print the last error message to `stderr`

Synopsis

```
#include <stdio.h>
#include <errno.h>  // only if you want to directly use the "errno" var

void perror(const char *s);
```

Description

Many functions, when they encounter an error condition for whatever reason, will set a global variable called `errno` (in `<errno.h>`) for you. `errno` is just an interger representing a unique error.

But to you, the user, some number isn't generally very useful. For this reason, you can call `perror()` after an error occurs to print what error has actually happened in a nice human-readable string.

And to help you along, you can pass a parameter, `s`, that will be prepended to the error string for you.

One more clever trick you can do is check the value of the `errno` (you have to include `errno.h` to see it) for specific errors and have your code do different things. Perhaps you want to ignore certain errors but not others, for instance.

The standard only defines three values for `errno`, but your system undoubtedly defines more. The three that are defined are:

errno	Description
EDOM	Math operation outside domain.
EILSEQ	Invalid sequence in multibyte to wide character encoding.
ERANGE	Result of operation doesn't fit in specified type.

The catch is that different systems define different values for `errno`, so it's not very portable beyond the above 3. The good news is that at least the values are *largely* portable between Unix-like systems, at least.

Return Value

Returns nothing at all! Sorry!

Example

`fseek()` returns `-1` on error, and sets `errno`, so let's use it. Seeking on `stdin` makes no sense, so it should generate an error:

```
#include <stdio.h>
#include <errno.h> // must include this to see "errno" in this example

int main(void)
{
    if (fseek(stdin, 10L, SEEK_SET) < 0)
        perror("fseek");

    fclose(stdin); // stop using this stream

```

```
    if (fseek(stdin, 20L, SEEK_CUR) < 0) {

        // specifically check errno to see what kind of
        // error happened...this works on Linux, but your
        // mileage may vary on other systems!

        if (errno == EBADF) {
            perror("fseek again, EBADF");
        } else {
            perror("fseek again");
        }
    }
}
```

And the output is:

```
fseek: Illegal seek
fseek again, EBADF: Bad file descriptor
```

See Also

`feof()`, `ferror()`,

`<string.h>` String Manipulation

As has been mentioned earlier in the guide, a string in C is a sequence of bytes in memory, terminated by a NUL character (‘`\0`’). The NUL at the end is important, since it lets all these string functions (and `printf()` and `puts()` and everything else that deals with a string) know where the end of the string actually is.

Fortunately, when you operate on a string using one of these many functions available to you, they add the NUL terminator on for you, so you actually rarely have to keep track of it yourself. (Sometimes you do, especially if you’re building a string from scratch a character at a time or something.)

In this section you’ll find functions for pulling substrings out of strings, concatenating strings together, getting the length of a string, and so forth and so on.

memcpy(), memmove()

Copy bytes of memory from one location to another

Synopsis

```
#include <string.h>

void *memcpy(void * restrict s1, const void * restrict s2, size_t n);

void *memmove(void *s1, const void *s2, size_t n);
```

Description

These functions copy memory—as many bytes as you want! From source to destination!

The main difference between the two is that `memcpy()` cannot safely copy overlapping memory regions, whereas `memmove()` can.

On the one hand, I'm not sure why you'd want to ever use `memcpy()` instead of `memmove()`, but I'll bet it's possibly more performant.

The parameters are in a particular order: destination first, then source. I remember this order because it behaves like an "=" assignment: the destination is on the left.

Return Value

Both functions return whatever you passed in for parameter `s1` for your convenience.

Example

```
char s[100] = "Goats";
char t[100];

memcpy(t, s, 6);       // Copy non-overlapping memory

memmove(s + 2, s, 6);  // Copy overlapping memory
```

See Also

`strcpy()`, `strncpy()`

strcpy(), strncpy()

Copy a string

Synopsis

```
#include <string.h>

char *strcpy(char *dest, char *src);

char *strncpy(char *dest, char *src, size_t n);
```

Description

These functions copy a string from one address to another, stopping at the NUL terminator on the `src`string.

`strncpy()` is just like `strcpy()`, except only the first `n` characters are actually copied. Beware that if you hit the limit, `n` before you get a NUL terminator on the `src` string, your `dest` string won't be NUL-terminated. Beware! BEWARE!

(If the `src` string has fewer than `n` characters, it works just like `strcpy()`.)

You can terminate the string yourself by sticking the `'\0'` in there yourself:

```
char s[10];
char foo = "My hovercraft is full of eels."; // more than 10 chars

strncpy(s, foo, 9); // only copy 9 chars into positions 0-8
s[9] = '\0';        // position 9 gets the terminator
```

Return Value

Both functions return `dest` for your convenience, at no extra charge.

Example

```
char *src = "hockey hockey hockey hockey hockey hockey hockey hockey";
char dest[20];

int len;

strcpy(dest, "I like "); // dest is now "I like "

len = strlen(dest);

// tricky, but let's use some pointer arithmetic and math to append
// as much of src as possible onto the end of dest, -1 on the length to
// leave room for the terminator:
strncpy(dest+len, src, sizeof(dest)-len-1);

// remember that sizeof() returns the size of the array in bytes
// and a char is a byte:
dest[sizeof(dest)-1] = '\0'; // terminate

// dest is now:       v null terminator
```

```
// I like hockey hocke
// 0123456789012345678901234 5
```

See Also

`memcpy()`, `strcat()`, `strncat()`

strcat(), strncat()

Concatenate two strings into a single string.

Synopsis

```
#include <string.h>

int strcat(const char *dest, const char *src);

int strncat(const char *dest, const char *src, size_t n);
```

Description

"Concatenate", for those not in the know, means to "stick together". These functions take two strings, and stick them together, storing the result in the first string.

These functions don't take the size of the first string into account when it does the concatenation. What this means in practical terms is that you can try to stick a 2 megabyte string into a 10 byte space. This will lead to unintended consequences, unless you intended to lead to unintended consequences, in which case it will lead to intended unintended consequences.

Technical banter aside, your boss and/or professor will be irate.

If you want to make sure you don't overrun the first string, be sure to check the lengths of the strings first and use some highly technical subtraction to make sure things fit.

You can actually only concatenate the first `n` characters of the second string by using `strncat()` and specifying the maximum number of characters to copy.

Return Value

Both functions return a pointer to the destination string, like most of the string-oriented functions.

Example

```
char dest[20] = "Hello";
char *src = ", World!";
char numbers[] = "12345678";

printf("dest before strcat: \"%s\"\n", dest); // "Hello"

strcat(dest, src);
printf("dest after strcat:  \"%s\"\n", dest); // "Hello, world!"

strncat(dest, numbers, 3); // strcat first 3 chars of numbers
printf("dest after strncat: \"%s\"\n", dest); // "Hello, world!123"
```

Notice I mixed and matched pointer and array notation there with `src` and `numbers`; this is just fine with string functions.

See Also

`strlen()`

`strcmp()`, `strncmp()`, `memcpy()`

Compare two strings or memory regions and return a difference.

Synopsis

```
#include <string.h>

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

int memcmp(const void *s1, const void *s2, size_t n);
```

Description

All these functions compare chunks of bytes in memory.

`strcmp()` and `strncmp()` operate on NUL-terminated strings, whereas `memcmp()` will compare the number of bytes you specify, brazenly ignoring any NUL characters it finds along the way.

`strcmp()` compares the entire string down to the end, while `strncmp()` only compares the first `n` characters of the strings.

It's a little funky what they return. Basically it's a difference of the strings, so if the strings are the same, it'll return zero (since the difference is zero). It'll return non-zero if the strings differ; basically it will find the first mismatched character and return less-than zero if that character in `s1` is less than the corresponding character in `s2`. It'll return greater-than zero if that character in `s1` is greater than that in `s2`.

So if they return `0`, the comparison was equal (i.e. the difference was `0`.)

These functions can be used as comparison functions for `qsort()` if you have an array of `char*`s you want to sort.

Return Value

Returns zero if the strings or memory are the same, less-than zero if the first different character in `s1` is less than that in `s2`, or greater-than zero if the first difference character in `s1` is greater than than in `s2`.

Example

```
char *s1 = "Muffin";
char *s2 = "Muffin Sandwich";
char *s3 = "Muffin";

strcmp("Biscuits", "Kittens"); // returns < 0 since 'B' < 'K'
strcmp("Kittens", "Biscuits"); // returns > 0 since 'K' > 'B'

if (strcmp(s1, s2) == 0)
    printf("This won't get printed because the strings differ");

if (strcmp(s1, s3) == 0)
    printf("This will print because s1 and s3 are the same");

// this is a little weird...but if the strings are the same, it'll
// return zero, which can also be thought of as "false". Not-false
// is "true", so (!strcmp()) will be true if the strings are the
```

```
// same. yes, it's odd, but you see this all the time in the wild
// so you might as well get used to it:

if (!strcmp(s1, s3))
    printf("The strings are the same!")

if (!strncmp(s1, s2, 6))
    printf("The first 6 characters of s1 and s2 are the same");
```

See Also

`memcmp()`, `qsort()`

`strcoll()`

Compare two strings accounting for locale

Synopsis

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Description

This is basically `strcmp()`, except that it handles accented characters better depending on the locale.

For example, my `strcmp()` reports that the character "é" (with accent) is greater than "f". But that's hardly useful for alphabetizing.

By setting the `LC_COLLATE` locale value (either by name or via `LC_ALL`), you can have `strcoll()` sort in a way that's more meaningful by the current locale. For example, by having "é" appear sanely *before* "f".

Return Value

Like the other string comparison functions, `strcoll()` returns a negative value if `s1` is less than `s2`, or a positive value if `s1` is greater than `s2`. Or `0` if they are equal.

Example

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

int main(void)
{
    setlocale(LC_ALL, "");

    // If your source character set doesn't support "é" in a string
    // you can replace it with `\u00e9`, the Unicode code point
    // for "é".

    printf("%d\n", strcmp("é", "f"));   // Reports é > f, yuck.
    printf("%d\n", strcoll("é", "f"));  // Reports é < f, yay!
}
```

See Also

`strcmp()`

strxfrm()

Transform a string for comparing based on locale

Synopsis

```
#include <string.h>

size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

Description

This is a strange little function, so bear with me.

Firstly, if you haven't done so, get familiar with `strcoll()` because this is closely related to that.

OK! Now that you're back, you can think of `strxfrm()` as the first part of the `strcoll()` internals. Basically, `strcoll()` has to transform a string into a form that can be compared with `strcmp()`. And it does this with `strxfrm()` for both strings every time you call it.

`strxform()` takes string `s2` and transforms it (readies it for `strcmp()`) storing the result in `s1`. It writes no more than `n` bytes, protecting us from terrible buffer overflows.

But hang on—there's another mode! If you pass `NULL` for `s1` and `0` for `n`, it will return the number of bytes that the transformed string *would have* used[160]. This is useful if you need to allocate some space to hold the transformed string before you `strcmp()` it against another.

What I'm getting at, not to be too blunt, is that `strcoll()` is slow compared to `strcmp()`. It does a lot of extra work running `strxfrm()` on all its strings.

In fact, we can see how it works by writing our own like this:

```
int my_strcoll(char *s1, char *s2)
{
    // Use n = 0 to just get the lengths of the transformed strings
    int len1 = strxfrm(NULL, s1, 0) + 1;
    int len2 = strxfrm(NULL, s2, 0)  + 1;

    // Allocate enough room for each
    char *d1 = malloc(len1);
    char *d2 = malloc(len2);

    // Transform the strings for comparison
    strxfrm(d1, s1, len1);
    strxfrm(d2, s2, len2);

    // Compare the transformed strings
    int result = strcmp(d1, d2);

    // Free up the transformed strings
    free(d2);
    free(d1);

    return result;
}
```

[160] It always returns the number of bytes the transformed string took, but in this case because `s1` was `NULL`, it doesn't actually write a transformed string.

You see on lines 12, 13, and 16, above how we transform the two input strings and then call `strcmp()` on the result.

So why do we have this function? Can't we just call `strcoll()` and be done with it?

The idea is that if you have one string that you're going to be comparing against a whole lot of other ones, maybe you just want to transform that string one time, then use the faster `strcmp()` saving yourself a bunch of the work we had to do in the function, above.

We'll do that in the example.

Return Value

Returns the number of bytes in the transformed sequence. If the value is greater than `n`, the results in `s1` are meaningless.

Example

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

// Transform a string for comparison, returning a malloc'd
// result
char *get_xfrm_str(char *s)
{
    int len = strxfrm(NULL, s, 0) + 1;
    char *d = malloc(len);

    strxfrm(d, s, len);

    return d;
}

// Does half the work of a regular strcoll() because the second
// string arrives already transformed.
int half_strcoll(char *s1, char *s2_transformed)
{
    char *s1_transformed = get_xfrm_str(s1);

    int result = strcmp(s1_transformed, s2_transformed);

    free(s1_transformed);

    return result;
}

int main(void)
{
    setlocale(LC_ALL, "");

    // Pre-transform the string to compare against
    char *s = get_xfrm_str("éfg");

```

```
    // Repeatedly compare against "éfg"
    printf("%d\n", half_strcoll("fgh", s));  // "fgh" > "éfg"
    printf("%d\n", half_strcoll("àbc", s));  // "àbc" < "éfg"
    printf("%d\n", half_strcoll("ĥij", s));  // "ĥij" > "éfg"

    free(s);
}
```

See Also

strcoll()

`strchr()`, `strrchr()`

Find a character in a string.

Synopsis

```
#include <string.h>

char *strchr(char *str, int c);

char *strrchr(char *str, int c);
```

Description

The functions `strchr()` and `strrchr` find the first or last occurrence of a letter in a string, respectively. (The extra "r" in `strrchr()` stands for "reverse"–it looks starting at the end of the string and working backward.) Each function returns a pointer to the char in question, or `NULL` if the letter isn't found in the string.

Quite straightforward.

One thing you can do if you want to find the next occurrence of the letter after finding the first, is call the function again with the previous return value plus one. (Remember pointer arithmetic?) Or minus one if you're looking in reverse. Don't accidentally go off the end of the string!

Return Value

Returns a pointer to the occurrence of the letter in the string, or `NULL` if the letter is not found.

Example

```
// "Hello, world!"
//       ^   ^
//       A   B

char *str = "Hello, world!";
char *p;

p = strchr(str, ','); // p now points at position A
p = strrchr(str, 'o'); // p now points at position B
```

```
// repeatedly find all occurrences of the letter 'B'
char *str = "A BIG BROWN BAT BIT BEEJ";
char *p;

for(p = strchr(str, 'B'); p != NULL; p = strchr(p + 1, 'B')) {
    printf("Found a 'B' here: %s\n", p);
}

// output is:
//
// Found a 'B' here: BIG BROWN BAT BIT BEEJ
// Found a 'B' here: BROWN BAT BIT BEEJ
// Found a 'B' here: BAT BIT BEEJ
// Found a 'B' here: BIT BEEJ
// Found a 'B' here: BEEJ
```

See Also

strspn(), strcspn()

Return the length of a string consisting entirely of a set of characters, or of not a set of characters.

Synopsis

```
#include <string.h>

size_t strspn(char *str, const char *accept);

size_t strcspn(char *str, const char *reject);
```

Description

`strspn()` will tell you the length of a string consisting entirely of the set of characters in `accept`. That is, it starts walking down `str` until it finds a character that is *not* in the set (that is, a character that is not to be accepted), and returns the length of the string so far.

`strcspn()` works much the same way, except that it walks down `str` until it finds a character in the `reject` set (that is, a character that is to be rejected.) It then returns the length of the string so far.

Return Value

The length of the string consisting of all characters in `accept` (for `strspn()`), or the length of the string consisting of all characters except `reject` (for `strcspn()`).

Example

```
char str1[] = "a banana";
char str2[] = "the bolivian navy on manuvers in the south pacific";

// how many letters in str1 until we reach something that's not a vowel?
n = strspn(str1, "aeiou");  // n == 1, just "a"

// how many letters in str1 until we reach something that's not a, b,
// or space?
n = strspn(str1, "ab "); // n == 4, "a ba"

// how many letters in str2 before we get a "y"?
n = strcspn(str2, "y"); // n = 16, "the bolivian nav"
```

See Also

`strchr()`, `strrchr()`

strstr()

Find a string in another string.

Synopsis

```
#include <string.h>

char *strstr(const char *str, const char *substr);
```

Description

Let's say you have a big long string, and you want to find a word, or whatever substring strikes your fancy, inside the first string. Then `strstr()` is for you! It'll return a pointer to the `substr` within the `str`!

Return Value

You get back a pointer to the occurrence of the `substr` inside the `str`, or `NULL` if the substring can't be found.

Example

```
char *str = "The quick brown fox jumped over the lazy dogs.";
char *p;

p = strstr(str, "lazy");
printf("%s\n", p); // "lazy dogs."

// p is NULL after this, since the string "wombat" isn't in str:
p = strstr(str, "wombat");
```

See Also

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

strtok()

Tokenize a string.

Synopsis

```
#include <string.h>

char *strtok(char *str, const char *delim);
```

Description

If you have a string that has a bunch of separators in it, and you want to break that string up into individual pieces, this function can do it for you.

The usage is a little bit weird, but at least whenever you see the function in the wild, it's consistently weird.

Basically, the first time you call it, you pass the string, `str` that you want to break up in as the first argument. For each subsequent call to get more tokens out of the string, you pass `NULL`. This is a little weird, but `strtok()` remembers the string you originally passed in, and continues to strip tokens off for you.

Note that it does this by actually putting a NUL terminator after the token, and then returning a pointer to the start of the token. So the original string you pass in is destroyed, as it were. If you need to preserve the string, be sure to pass a copy of it to `strtok()` so the original isn't destroyed.

Return Value

A pointer to the next token. If you're out of tokens, `NULL` is returned.

Example

```
// break up the string into a series of space or
// punctuation-separated words
char *str = "Where is my bacon, dude?";
char *token;

// Note that the following if-do-while construct is very very
// very very very common to see when using strtok().

// grab the first token (making sure there is a first token!)
if ((token = strtok(str, ".,?! ")) != NULL) {
    do {
        printf("Word: \"%s\"\n", token);

        // now, the while continuation condition grabs the
        // next token (by passing NULL as the first param)
        // and continues if the token's not NULL:
    } while ((token = strtok(NULL, ".,?! ")) != NULL);
}

// output is:
//
// Word: "Where"
// Word: "is"
// Word: "my"
// Word: "bacon"
```

```
// Word: "dude"
//
```

See Also

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

`strlen()`

Returns the length of a string.

Synopsis

```
#include <string.h>

size_t strlen(const char *s);
```

Description

This function returns the length of the passed null-terminated string (not counting the NUL character at the end). It does this by walking down the string and counting the bytes until the NUL character, so it's a little time consuming. If you have to get the length of the same string repeatedly, save it off in a variable somewhere.

Return Value

Returns the number of bytes in the string. Note that this might be different than the number of characters in a multibyte string.

Example

```
char *s = "Hello, world!"; // 13 characters

// prints "The string is 13 characters long.":

printf("The string is %d characters long.\n", strlen(s));
```

See Also

Mathematics

It's your favorite subject: Mathematics! Hello, I'm Doctor Math, and I'll be making math FUN and EASY!

[vomiting sounds]

Ok, I know math isn't the grandest thing for some of you out there, but these are merely functions that quickly and easily do math you either know, want, or just don't care about. That pretty much covers it.

For you trig fans out there, we've got all manner of things, including sine, cosine, tangent, and, conversely, arc sine, arc cosine, and arc tangent. That's very exciting.

And for normal people, there is a slurry of your run-of-the-mill functions that will serve your general purpose mathematical needs, including absolute value, hypotenuse length, square root, cube root, and power.

In short, you're a fricking MATHEMATICAL DEITY!

Oh wait, before then, I should tell you that the trig functions have three variants with different suffixes. The "f" suffix (e.g. `sinf()`) returns a `float`, while the "l" suffix (e.g. `sinl()`) returns a massive and nicely accurate `long double`. Normal `sin()` just returns a `double`. These are extensions to ANSI C, but they should be supported by modern compilers.

Also, there are several values that are defined in the `math.h` header file.

Constant	C Macro Equivalent
e	`M_E`
$\log_2 e$	`M_LOG2E`
$\log_{10} e$	`M_LOG10E`
$\log_e 2$	`M_LN2`
$\log_e 10$	`M_LN10`
π	`M_PI`
$\pi/2$	`M_PI_2`
$\pi/4$	`M_PI_4`
$1/\pi$	`M_1_PI`
$2/\pi$	`M_2_PI`
$2/\sqrt{\pi}$	`M_2_SQRTPI`
$\sqrt{2}$	`M_SQRT2`
$1/\sqrt{2}$	`M_SQRT1_2`

`sin()`, `sinf()`, `sinl()`

Calculate the sine of a number.

Synopsis

```
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Description

Calculates the sine of the value `x`, where `x` is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the sine of `x`. The variants return different types.

Example

```
double sinx;
long double ldsinx;

sinx = sin(3490.0); // round and round we go!
ldsinx = sinl((long double)3.490);
```

See Also

`cos()`, `tan()`, `asin()`

cos(), cosf(), cosl()

Calculate the cosine of a number.

Synopsis

```
#include <math.h>

double cos(double x)
float cosf(float x)
long double cosl(long double x)
```

Description

Calculates the cosine of the value x, where x is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the cosine of x. The variants return different types.

Example

```
double sinx;
long double ldsinx;

sinx = sin(3490.0); // round and round we go!
ldsinx = sinl((long double)3.490);
```

See Also

sin(), tan(), acos()

tan(), tanf(), tanl()

Calculate the tangent of a number.

Synopsis

```
#include <math.h>

double tan(double x)
float tanf(float x)
long double tanl(long double x)
```

Description

Calculates the tangent of the value x, where x is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the tangent of x. The variants return different types.

Example

```
double tanx;
long double ldtanx;

tanx = tan(3490.0); // round and round we go!
ldtanx = tanl((long double)3.490);
```

See Also

sin(), cos(), atan(), atan2()

asin(), asinf(), asinl()

Calculate the arc sine of a number.

Synopsis

```
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Description

Calculates the arc sine of a number in radians. (That is, the value whose sine is `x`.) The number must be in the range -1.0 to 1.0.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the arc sine of `x`, unless `x` is out of range. In that case, `errno` will be set to EDOM and the return value will be NaN. The variants return different types.

Example

```
double asinx;
long double ldasinx;

asinx = asin(0.2);
ldasinx = asinl((long double)0.3);
```

See Also

`acos()`, `atan()`, `atan2()`, `sin()`

acos(), acosf(), acosl()

Calculate the arc cosine of a number.

Synopsis

```
#include <math.h>

double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Description

Calculates the arc cosine of a number in radians. (That is, the value whose cosine is `x`.) The number must be in the range -1.0 to 1.0.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the arc cosine of `x`, unless `x` is out of range. In that case, `errno` will be set to EDOM and the return value will be NaN. The variants return different types.

Example

```
double acosx;
long double ldacosx;

acosx = acos(0.2);
ldacosx = acosl((long double)0.3);
```

See Also

`asin()`, `atan()`, `atan2()`, `cos()`

`atan()`, `atanf()`, `atanl()`,

`atan2()`, `atan2f()`, `atan2l()`

Calculate the arc tangent of a number.

Synopsis

```
#include <math.h>

double atan(double x);
float atanf(float x);
long double atanl(long double x);

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Description

Calculates the arc tangent of a number in radians. (That is, the value whose tangent is `x`.)

The `atan2()` variants are pretty much the same as using `atan()` with `y`/`x` as the argument…except that `atan2()` will use those values to determine the correct quadrant of the result.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

The `atan()` functions return the arc tangent of `x`, which will be between PI/2 and -PI/2. The `atan2()` functions return an angle between PI and -PI.

Example

```
double atanx;
long double ldatanx;

atanx = atan(0.2);
ldatanx = atanl((long double)0.3);

atanx = atan2(0.2);
ldatanx = atan2l((long double)0.3);
```

See Also

`tan()`, `asin()`, `atan()`

sqrt()

Calculate the square root of a number

Synopsis

```
#include <math.h>

double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Description

Computes the square root of a number. To those of you who don't know what a square root is, I'm not going to explain. Suffice it to say, the square root of a number delivers a value that when squared (multiplied by itself) results in the original number.

Ok, fine—I did explain it after all, but only because I wanted to show off. It's not like I'm giving you examples or anything, such as the square root of nine is three, because when you multiply three by three you get nine, or anything like that. No examples. I hate examples!

And I suppose you wanted some actual practical information here as well. You can see the usual trio of functions here—they all compute square root, but they take different types as arguments. Pretty straightforward, really.

Return Value

Returns (and I know this must be something of a surprise to you) the square root of `x`. If you try to be smart and pass a negative number in for `x`, the global variable `errno` will be set to `EDOM` (which stands for DOMain Error, not some kind of cheese.)

Example

```
// example usage of sqrt()

float something = 10;

double x1 = 8.2, y1 = -5.4;
double x2 = 3.8, y2 = 34.9;
double dx, dy;

printf("square root of 10 is %.2f\n", sqrtf(something));

dx = x2 - x1;
dy = y2 - y1;
printf("distance between points (x1, y1) and (x2, y2): %.2f\n",
    sqrt(dx*dx + dy*dy));
```

And the output is:

```
square root of 10 is 3.16
distance between points (x1, y1) and (x2, y2): 40.54
```

See Also

`hypot()`