Beej's Guide to C Programming

Brian “Beej” Hall
beej@beej.us

Revision alpha-25
May 17, 2007

Contents

1. 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 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.

1.1. Audience

As with most Beej's Guides, this one tries to cater to people who are just starting on the topic. That's you! 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?

1.2. Platform and Compiler

I'll try to stick to Good Ol'-Fashioned ANSI C, just like Mom used to bake. Well, for the most part. Here and there I talk about things that are in subsequent C standards, just to try to keep up to date.

My compiler of choice is GNU **gcc** since that's available on most systems, including the Linux systems on which I work.

Since the code is basically standard, it should build with virtually any C compiler on virtually any platform. If you're using Windows, run the result in a DOS window. All sample code will be using the console (that's “text window” for you kids out there), except for the sample code that doesn't.

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. Some of the compilers I am familiar with are the following:

- **GCC**[2]: GNU's C compiler, available for almost every platform, and popularly installed on Unix machines.
- **Digital Mars C/C++**[3]: The hackers at Digital Mars have a pretty rippin' C/C++ compiler for Windows that you can download and use for free, and that will work wonderfully for all the code presented in this guide. I highly recommend it.

1. `http://www.ioccc.org/`

- **VC++**[4]: Microsoft's Visual C++ for Windows. This is the standard that most Microsoft programmers use, and I freaking hate it. Nothing personal, but I'm one of those crazy people that still uses **vi**.
- **Turbo C**[5]: This is a classic compiler for MSDOS. It's downloadable for free, and I has a special place in my heart. (It can't handle the "`//`"-style comments, so they should all be converted to "`/**/`"-style.)
- **`cc`**: Virtually every Unix system has a C compiler installed, and they're typically and merely named `cc` (C Compiler, see?) Just try it from the command line and see what happens!

1.3. Building under Unix

If you have a source file called *`foo.c`*, it can be built with the following command from the shell:

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

This tells the compiler to build *`foo.c`*, and output an executable called *`foo`*. If **gcc** doesn't work, try using just **cc** instead.

1.4. Official Homepage

This official location of this document is http://beej.us/guide/bgc/[6]. Maybe this'll change in the future, but it's more likely that all the other guides are migrated off Chico State computers.

1.5. Email Policy

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

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. For more pointers, read ESR's document, How To Ask Questions The Smart Way[7].

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

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

1.6. Mirroring

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

1.7. Note for Translators

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

6. `http://beej.us/guide/bgc/`
7. `http://www.catb.org/~esr/faqs/smart-questions.html`

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

Sorry, but due to space constraints, I cannot host the translations myself.

1.8. Copyright and Distribution

2. Programming Building Blocks

"Where do these stairs go?"
"They go up."
--Ray Stantz and Peter Venkman, Ghostbusters

What is programming, anyway? I mean, you're learning how to do it, but what is it? Well, it's, umm, kind of like, well, say you have this multilayered chocolate and vanilla cake sitting on top of an internal combustion engine and the gearbox is connected to the coil with a banana. Now, if you're eating the cake a la mode, that means... Wait. Scratch that analogy. I'll start again.

What is programming, anyway? It's telling the computer how to perform a task. So you need two things (besides your own self and a computer) to get going. One thing you need is the task the computer is to perform. This is easy to get as a student because the teacher will hand you a sheet of paper with an assignment on it that describes exactly what the computer is to do for you to get a good grade.

If you don't want a good grade, the computer can do that without your intervention. But I digress.

The second thing you need to get started is the knowledge of how to tell the computer to do these things. It turns out there are lots of ways to get the computer to do a particular task...just like there are lots of ways to ask someone to please obtain for me my fluffy foot covering devices in order to prevent chilliness. Many of these ways are right, and a few of them are best.

What you can do as a programmer, though, is get through the assignments doing something that works, and then look back at it and see how you could have made it better or faster or more concise. This is one thing that seriously differentiates programmers from excellent programmers.

Eventually what you'll find is that the stuff you wrote back in college (e.g. The Internet Pizza Server, or, say, my entire Masters project) is a horridly embarrassing steaming pile of code that was quite possibly the worst thing you've ever written.

The only way to go is up.

2.1. The Specification

In the beginning was the plan
And then came the assumptions
And the assumptions were without form
And the plan was completely without substance
And the darkness was upon the face of workers
--Excerpt from The Plan, early Internet folklore

Ooooo! Prostrate yourself, mortal, in the face of The Specification!

Ok, maybe I'm being a little too overdramatic here. But I wanted to stress just mildly and subtly, if you might indulge me, that ***The Specification*** BWHAHAHA **THUNDERCLAP** (Sorry! Sorry!) is something you should spend time absorbing before your fingers touch the keyboard. Except for checking your mail and reading Slashdot, obviously. That goes without saying.

So what do you do with this specification? It's a description of what the program is going to do, right? But where to begin? What you need to do is this: break down the design into handy bite-sized pieces that you can implement using techniques you know work in those situations.

As you learn C, those bite-sized pieces will correspond to function calls or statements that you will have learned. As you learn to program in general, those bite-sized pieces will start corresponding to larger algorithms that you know (or can easily look up.)

Right now, you might not know any of the pieces that you have at your disposal. That's ok. The fastest way to learn them is to, right now, press the mouse to your forehead and say the password, “K&R2”.

That didn't work? Hmmm. There must be a problem with the system somewhere. Ok, we'll do it the old-school way: learning stuff by hand.

Let's have an example:

Assignment: Implement a program that will calculate the sum of all numbers between 1 and the number the user enters. The program shall output the result.

Ok, well, that summary is pretty high level and doesn't lend itself to bite-sized pieces, so it's up to us to split it up.

There are several places that are good to break up pieces to be more bite-sized. Input is one thing to break out, output is another. If you need to input something, or output something, each of those is a handy bite-sized piece. If you need to calculate something, that can be another bite-sized piece (though the more difficult calculations can be made up of many pieces themselves!)

So, moving forward through a sample run of the program:

1. We need the program to read a number from the keyboard.
2. We need the program to compute a result using that number.
3. We need the program to output the result.

This is good! We've identified the parts of the assignment that need to be worked on.

“Wait! Stop!” I hear you. You're wondering how we knew it was broken down into enough bite-sized pieces, and, in fact, how we even know those are bite-sized pieces, anyhow! For all you know, reading a number from the keyboard could be a hugely involved task!

The short of it is, well, you caught me trying to pull a fast one on you. *I* know these are bite-sized because in my head I can correspond them to simple C function calls or statements. Outputting the result, for instance, is one line of code (very bite-sized). But that's me and we're talking about you. In your case, I have a little bit of a chicken-and-egg problem: you need to know what the bite-sized pieces of the program are so you can use the right functions and statements, and you need to know what the functions and statements are in order to know how to split the project up into bite-sized pieces! Hell's bells!

So we're compromising a bit. I agree to tell you what the statements and functions are if *you* agree to keep this stuff about bite-sized pieces in the back of your head while we progress. Ok?

...I said, “Ok?” And you answer... “Ok, I promise to keep this bite-sized pieces stuff in mind.” Excellent!

2.2. The Implementation

Right! Let's take that example from the previous section and see how we're going to actually implement it. Remember that once you have the specification (or assignment, or whatever you're going to call it) broken up into handy bite-sized pieces, then you can start writing the instuctions to make that happen. Some bite-sized pieces might only have one statement; others might be pages of code.

Now here we're going to cheat a little bit again, and I'm going to tell you what you'll need to call to implement our sample program. I know this because I've done it all before and looked it all up. You, too, will soon know it for the same reasons. It just takes time and a lot of reading what's in the reference section of your C books.

So, let's take our steps, except this time, we'll write them with a little more information. Just bear with me through the syntax here and try to make the correlation between this and the bite-sized pieces mentioned earlier. All the weird parentheses and squirrely braces will make sense in later sections of the guide. Right now what's important is the steps and the translation of those steps to computer code.

The steps, partially translated:

1. Read the number from the keyboard using **`scanf()`**.
2. Compute the sum of the numbers between one and the entered number using a **`for`**-loop and the addition operator.
3. Print the result using **`printf()`**.

Normally, this partial translation would just take place in your head. You need to output to the console? You know that the **`printf()`** function is one way to do it.

And as the partial translation takes place in your head, what better time than that to actually code it up using your favorite editor:

```
#include <stdio.h>

int main(void)
{
    int num, result = 0;

    scanf("%d", &num); // read the number from the keyboard

    for(i = 1; i <= num; i++) { // compute the result
        result += i;
    }

    printf("%d\n", result); // output the result

    return 0;
}
```

Remember how I said there were multiple ways to do things? Well, I didn't have to use **`scanf()`**, I didn't have to use a **`for`**-loop, and I didn't have to use **`printf()`**. But they were the best for this example. `:-)`

2.3. So Much To Do, So Little Time

Another name for this section might have been, “Why can't I write a Photoshop clone in half an hour?”

Lots of people when they first start programming become disheartened because they've just spent all this effort to learn this whole programming business and what do they have to show for it: a little text-based program that prints a string that looks like it's some prehistoric throwback to 1979.

Well, I wish I could sugarcoat this a little bit more, but that is unfortunately the way it tends to go when you're starting out. Your first assignment is unlikely to be DOOM III, and is more likely to be something similar to:

```
Hello, I am the computer and I know that 2+2 =   4!
```

You elite coder, you.

Remember, though, how I said that eventually you'll learn to recognize larger and larger bite-sized pieces? What you'll eventually built up is a set of *libraries* (collections of reusable code) that you can use as building blocks for other programs.

For example, when I want to draw a bitmap on the screen, do I write a system to read the bytes of the file, decode the JPEG image format, detect video hardware, determine video memory layout, and copy the results onto the display? Well do I, punk? No. I call **`loadJPEG()`**. I call **`displayImage()`**. These are examples of functions that have already been written for my use. That makes it easy!

So you can plan ahead and figure out which components can be built up and reused, or you can use components that other people have built for you.

Examples pre-built components are: the standard C library (**`printf()`**, which we'll be using a lot of in this guide), the GTK+ library (a GUI library used with GNOME), the Qt toolkit (a GUI library used with the K Desktop), libSDL (a library for doing graphics), OpenGL (a library for doing 3D graphics), and so on, and so on. You can use all these to your own devious ends and you don't have to write them again!

2.4. Hello, World!

This is the canonical example of a C program. Everyone uses it:

```
/* helloworld program */

#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");

    return 0;
}
```

We're going to don our long-sleeved heavy-duty rubber gloves, grab a scapel, 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. 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.

(Modern C compilers also treat anything after a `//` as a comment. GCC will obey it, as will VC++. However, if you are using an old compiler like Turbo C, well, the `//` construct was a little bit before its time. So I'll try to keep it happy and use the old-style `/*comments*/` in my code. But everyone uses `//` these days when adding a comment to the end of a line, and you should feel free to, as well.)

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 gratuitiously 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, machine code, 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. Octothorp.

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 IO" (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.

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.

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

A *function*. "Great! More terminology I don't know!" I feel your pain, but can only offer you the cold heartless comfort of more words: a function is a collection of code that is to be executed as a group when the function is called. You can think of it as, "When I call **`main()`**, all the stuff in the squirrley braces will be executed, and not a moment before."

How do you call a 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 squirrley 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 squirrely 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, build it, and run it--see what happens:

```
Hello, World!
```

It's done and tested! Ship it!

3. Variables, Expressions, and Statements (Oh My)

“It takes all kinds to make a world, does it not, Padre?”
“So it does, my son, so it does.”
Pirate Captain Thomas Bartholomew Red to the Padre, Pirates

There sure can be lotsa stuff in a C program.

Yup.

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

3.1. Variables

A variable is simply a name for a number. The number associated with that variable is said to be it's *value*. You can change the value later, too. One minute a variable named `foo` might have the value 2, the next you might change it to 3. It's *variable*, see?

Variables can have different *types*, as well. In C, because it's such a picky language about types (I'm saying that emphatically to make strongly-typed language fanatics roll in their future graves) you have to know in advance what type of numbers you'll be storing in the variable.

Before you can use a variable in your code, you have to *declare* it. This way the compiler knows in advance as it runs down your program how to treat any variables. Here is an example that shows a number of different types, and their corresponding sets of numbers they represent:

```
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 */

    return 0;
}
```

In the above example, we've declared a couple of variables. We haven't used them yet, and they're both uninitialized. One holds an integer number (random, to start, since we never initialized it), and the other holds a floating point number (a real number, basically.)

What's this? You want to store some numbers in those variables? Well, ask your mother; if it's all right with her, it's all right with me.

So, how to do that...you use the *assignment* operator. An operator is generally a piece of punctuation that operates on two expressions to produce a result. There are a number of them, like the addition operator (+) and the subtraction operator (-), but I'm getting ahead of myself. We'll talk about those more in the expressions section, below. In this case, we're going to use the assignment operator, which is =, to assign a value into a variable:

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

    i = 2; /* assign the value 2 into the variable i */

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

    return 0;
```

```
}
```

Killer. We've stored a value. But don't run off an implement a clone of Quake III just yet; let's try to do something with that variable, just so we can say we did. Let's print it out using **`printf()`**, for instance.

We're going to do that by passing *two* 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 strings 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.

As such, we can print out the value of *`i`* like so:

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

    i = 2; /* assign the value 2 into the variable i */

    printf("Hello, World!  The value of i is %d, okay?\n", i);

    return 0;
}
```

And the output will be:

```
Hello, World!  The value of i is 2, okay?
```

And now, on to expressions! We'll do all kinds of fancy stuff with these variables.

3.2. Operators

I've snuck a few quick ones past you already when it comes to expressions. You've already seen things like:

```
result += i;
i = 2;
```

Those are both expressions. In fact, you'll come to find that most everything in C is an expression of one form or another. But here for the start I'll just tell you about a few common types of operators that you probably want to know about.

```
i = i + 3;  /* addition (+) and assignment (=) operators */
i = i - 8;  /* subtraction, subtract 8 from i */
i = i / 2;  /* division */
i = i * 9;  /* multiplication */
i++;        /* add one to i ("post-increment"; more later) */
++i;        /* add one to i ("pre-increment") */
i--;        /* subtract one from i ("post-decrement") */
```

Looks pretty weird, that `i = i + 3` expression, huh. I mean, it makes no sense algebraically, right? That's true, but it's because it's not really algebra. That's not an equivalency statement--it's an assignment. Basically it's saying whatever variable is on the left hand side of the assignment (=) is going to be assigned the value of the expression on the right. So it's all ok.

An expression? What's that? Sorry, but I'm far to lazy to cover it here, so I'll cover it in the next section.

3.3. Expressions

This is actually one of the most important sections in the guide about C++. So pay attention and be seated, you naughty children!

Now that everything's in order, we'll...actually, let's do the important part a little later. Now we'll talk about what those expression things are a little bit.

An expression in C consists of other expressions optionally put together with operators. I know that's a self-referential definition, but those are always good for a laugh, aren't they? (Pick up any old C book and look up *recursion* in the index and see what pages it leads you to.)

The basic building block expressions that you put together with operators are variables, constant numbers (like 10 or 12.34), and functions. (We'll talk about functions later, although if you recall we've already had a run-in with the **`printf()`** function.)

And so when you chain these together with operators, the result is an expression, as well. All of the following are valid C expressions:

```
i = 3
i++
i = i + 12
i + 12
2
f += 3.14
```

Now where can you use these expressions? Well, you can use them in a function call (I know, I know--I'll get to function real soon now), or as the right hand side of an assignment. You have to be more careful with the left side of an assignment; you can only use certain things there, but for now suffice it to say it must be a single variable on the left side of the assignment, and not a complicated expression:

```
radius = circumference / (2.0 * 3.14159);       /* valid */
diameter / 2 = circumference / (2.0 * 3.14159);  /* INVALID */
```

(I also slipped more operator stuff in there--the parentheses. These cause part of an expression (yes, any old expression) to be evaluated first by the compiler, before more of the expression is computed. This is just like parentheses work in algebra.)

Well, I was going to talk about that important thing in this section, but now it looks like we're running behind a bit, so let's postpone it until later. I promise to catch up, don't you fear!

3.4. Statements

For the most part, you're free to make up variable names to your heart's content, calling them whatever you'd like. There are no exceptions, except for statements and other *reserved words* which you may not use unless you use them in the officially (by the compiler) prescribed manner.

That definition was rather glib. I should rewrite it. Or maybe we'll just sojourn bravely on! Yes!

What are these pesky statements? Let's say, completely hypothetically, you want to do something more than the already amazingly grand example program of assigning a value to a variable and printing it. What if you only want to print it if the number is less than 10? What if you want to print all numbers between it and 10? What if you want to only print the number on a Thursday? All these incredible things and more are available to you through the magic of various statements.

3.4.1. The if statement

The easiest one to wrap your head around is the conditional statement, if. It can be used to do (or not do) something based on a condition.

Like what kind of condition? Well, like is a number greater than 10?

```
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) print ("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.

What are the conditions?

```
i == 10;    /* true if i is equal to 10 */
i != 10;    /* true if i is not equal to 10 */
i > 10;     /* true if i greater than 10 */
i < 10;     /* true if i less than 10 */
i >= 10;   /* true if i greater than  or equal to 10 */
i <= 10;   /* true if i less than or equal to 10 */
i <= 10;   /* true if i less than or equal to 10 */
```

Guess what these all are? No really, guess. They're expressions! Just like before! So statements take an expression (some statements take multiple expressions) and evaluate them. The if statement evaluates to see if the expression is true, and then executes the following code if it is.

What is “true” anyway? C doesn't have a “true” keyword like C++ does. In C, any non-zero value is true, and a zero value is false. For instance:

```
if (1)   printf("This will always print.\n");
if (-3490) printf("This will always print.\n");
if (0)   printf("This will never print.  Ever.\n");
```

And the following will print `1` followed by `0`:

```
int i = 10;

printf("%d\n", i == 10); /* i == 10 is true, so it's 1 */
printf("%d\n", i > 20);  /* i is not > 20, so this is false, 0 */
```

(Hey, look! We just passed those expressions as arguments to the function **`printf()`**! Just like we said we were going to do before!)

Now, one common pitfall here with conditionals is that you end up confusing the assignment operator (=) with the comparison operator (==). Note that the results of both operators is an expression, so both are valid to put inside the if statement. Except one assigns and the other compares! You most likely want to compare. If weird stuff is happening, make sure you have the two equal signs in your comparison operator.

3.4.2. The while statement

Let's have another statement. Let's say you want to repeatly perform a task until a condition is true. This sounds like a job for the while loop. This works just like the if statement, except that it will repeately execute the following block of code until the statement is false, much like an insane android bent on killing its innocent masters.

Or something.

Here's an example of a while loop that should clarify this up a bit and help cleanse your mind of the killing android image:

```
// 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");
```

The easiest way to see what happens here is to mentally step through the code a line at a time.

1. First, *`i`* is set to zero. It's good to have these things initialized.
2. Secondly, we hit the while statement. It checks to see if the *continuation condition* is true, and continues to run the following block if it is. (Remember, true is `1`, and so when *`i`* is zero, the expression `i < 10` is `1` (true).
3. Since the continuation condition was true, we get into the block of code. The **`printf()`** function executes and outputs "`i is now 0!`".
4. Next, we get that post-increment operator! Remember what it does? It adds one to *`i`* in this case. (I'm going to tell you a little secret about post-increment: the increment happens *AFTER all of the rest of the expression has been evaluated*. That's why it's called "post", of course! In this case, the entire expression consists of simply *`i`*, so the result here is to simply increment *`i`*.
5. Ok, now we're at the end of the basic block. Since it started with a while statement, we're going to loop back up to the while and then:
6. We have arrived back at the start of the while statement. It seems like such a long time ago we were once here, doesn't it? Only this time things seem slightly different...what could it be? A new haircut, perhaps? No! The variable *`i`* is equal to `1` this time instead of `0`! So we have to check the continuation condition again. Sure enough, `1 < 10` last time I checked, so we enter the block of code again.
7. We **`printf()`** "`i is now 1!`".
8. We increment *`i`* and it goes to `2`.
9. We loop back up to the while statement and check to see if the continuation condition is true.
10. Yadda, yadda, yadda. I think you can see where this is going. Let's skip ahead to the *incredible future* where people commute to work on their AI-controlled rocket scooters, eat anti-gravity beets, and little spherical robot helicopters freely roam the skies making *beep-beep-beep-beep* noises. And where the variable *`i`* has finally

been incremented so it's value is `10`. Meanwhile, while we've slept in cryogenic hybernation, our program has been dutifully fulfilling its thousand-year mission to print things like "i is now 4!", "i is now 5!", and finally, "i is now 9!"

11. So *`i`* has finally been incremented to `10`, and we check the continuation condition. It `10 < 10`? Nope, that'll be false and zero, so the while statement is finally completed and we continue to the next line.
12. And lastly **`printf`** is called, and we get our parting message: "`All done!`".

That was a lot of tracing, there, wasn't it? This kind of mentally running through a program is commonly called *desk-checking* your code, because historically you do it sitting at your desk. It's a powerful debugging technique you have at your disposal, as well.

3.4.3. The do-while statement

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

They are basically the same, except if the continuation 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 continuation 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 continuation condition is false right away. So in the while, the condition 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 continuation condition, use do-while.

3.4.4. The for statement

Now you're starting to feel more comfortable with these looping statements, huh! Well, listen up! It's time for something a little more complicated: the for statement. This is another looping construct that gives you a cleaner syntax than while in many cases, but does basically the same thing. Here are two pieces of equivalent code:

```
// using a while statement:

// print numbers between 0 and 9, inclusive:
i = 0;
while (i < 10) {
    printf("i is %d\n");
    i++;
}

// do the same thing with a for-loop:
for (i = 0; i < 10; i++) {
    printf("i is %d\n");
}
```

That's right, kids--they do exactly the same thing. But you can see how the for statement is a little more compact and easy on the eyes.

It's split into three parts, separated by semicolons. The first is the initialization, the second is the continuation condition, and the third is what should happen at the end of the block if the contination condition is true. All three of these parts are optional. And empty for will run forever:

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

}
```

4. Building Blocks Revisited

"Is everything all right, sir?"
"No. No, it's not. Some smegger's filled out this 'Have You Got a Good Memory' quiz."
"Why, that was you, sir. Don't you remember?"
--Kryten and Dave Lister, Red Dwarf

Before we start with functions in the next section, we're going to quickly tie this in with that very important thing to remember back at the beginning of the guide. Now what was it...oh, well, I guess I gave it away with this section title, but let's keep talking as if that didn't happen.

Yes, it was basic building blocks, and how you take a specification and turn it into little bite-sized pieces that you can easily translate into blocks of code. I told you to take it on faith that I'd tell you some of the basic pieces, and I'm just reminding you here, in case you didn't notice, that all those statements back there are little basic building blocks that you can use in your programs.

Such as, if the specification reads:

Assignment: Write a program that repeatedly accepts user input and then prints the numbers between 0 and the entered number. If the user enters a number less than or equal to zero, the program will exit.

You now have enough information to figure out all the basic building blocks to make this program happen. You'll have to steal **`scanf()`** and **`printf()`** usage from previous examples, but the rest of the parts correspond to various statements you've seen in the previous section.

Note that off the top of my head, I can think of many many ways to implement this assignment. If you come up with one that works, good for you! Sure, it might not be the best, but what is "best", anyway? (Well, it turns out best is defined as what your professor or boss thinks is best, but let's be happily theoretical for the moment. Ahhh.)

So! Do some breakdown of the above assignment, and come up with the basic structure you'd use. In fact, go ahead and code it up, and try it out!

5. Functions

With the previous section on building blocks fresh in your head, let's imagine a freaky world where a program is so complicated, so insidiously large, that once you shove it all into your **`main()`**, it becomes rather unwieldy.

What do I mean by that? The best analogy I can think of is that programs are best read, modified, and understood by humans when they are split into convenient pieces, like a book is most conveniently read when it is split into paragraphs.

Ever try to read a book with no paragraph breaks? It's tough, man, believe me. I once read through *Captain Singleton* by Daniel Defoe since I was a fan of his, but Lord help me, the man didn't put a single paragraph break in there. It was a brutal novel.

But I digress. What we're going to do to help us along is to put some of those building blocks in their own functions when they become too large, or when they do a different thing than the rest of the code. For instance, the assignment might call for a number to be read, then the sum of all number between 1 and it calculated and printed. It would make sense to put the code for calculating the sum into a separate function so that the main code a) looks cleaner, and b) the function can be *reused* elsewhere.

Reuse is one of the main reasons for having a function. Take the **`printf()`** for instance. It's pretty complicated down there, parsing the format string and knowing how to actually output characters to a device and all that. Imagine if you have to rewrite all that code every single time you wanted to output a measly string to the console? No, no--far better to put the whole thing in a function and let you just call it repeatedly, see?

You've already seen a few functions called, and you've even seen one *defined*, namely the almighty **`main()`** (the definition is where you actually put the code that does the work of the function.) But the **`main()`** is a little bit incomplete in terms of how it is defined, and this is allowed for purely historical reasons. More on that later. Here we'll define and call a normal function called **`plus_one()`** that take an integer parameter and returns the value plus one:

```
int plus_one(int n) /* THE DEFINITION */
{
    return n + 1;
}

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

    j = plus_one(i); /* THE CALL */

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

    return 0;
}
```

(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.)

So here we have a function definition for **`plus_one()`**. The first word, `int`, is the return type of the function. That is, when we come back to use it in **`main()`**, the value of the expression (in the case of the call, the expression is merely the call itself) will be of this type.

By wonderful coincidence, you'll notice that the type of *j*, the variable that is to hold the return value of the function, is of the same type, `int`. This is completely on purpose.

Then we have the function name, followed by a *parameter list* in parenthesis. These correspond to the values in parenthesis in the call to the function...but *they don't have to have the same names*. Notice we call it with *i*, but the variable in the function definition is named *n*. This is ok, since the compiler will keep track of it for you.

Inside the **`plus_one()`** itself, we're doing a couple things on one line here. We have an expression `n + 1` that is evaluated before the return statement. So if we pass the value 10 into the function, it will evaluate `10 + 1`, which, in this universe, comes to 11, and it will return that.

Once returned to the call back in **`main()`**, we do the assignment into *j*, and it takes on the return value, which was 11. Hey look! *j* is now *i* plus one! Just like the function was supposed to do! This calls for a celebration!

[*GENERIC PARTY SOUNDS.*]

Ok, that's enough of that. Now, a couple paragraphs back, I mentioned that the names in the parameter list of the function definition correspond to the *values* passed into the function. In the case of **`plus_one()`**, you can call it any way you like, as long as you call it with an `int`-type parameter. For example, all these calls are valid:

```
int a = 5, b = 10;

plus_one(a);     /* the type of a is int */
plus_one(10);   /* the type of 10 is int */
plus_one(1+10);  /* the type of the whole expression is still int */
plus_one(a+10);  /* the type of the whole expression is still int */
plus_one(a+b);   /* the type of the whole expression is still int */
plus_one(plus_one(a));  /* oooo! return value is int, so it's ok!  */
```

If you're having trouble wrapping your head around that last line there, just take it one expression at a time, starting at the innermost parentheses (because the innermost parentheses are evaluated first, rememeber?) So you start at *a* and think, that's a valid `int` to call the function **`plus_one()`** with, so we call it, and that returns an `int`, and that's a valid type to call the next outer **`plus_one()`** with, so we're golden.

Hey! What about the return value from all of these? We're not assigning it into anything! Where is it going? Well, on the last line, the innermost call to **`plus_one()`** is being used to call **`plus_one()`** again, but aside from that, you're right--they are being discarded completely. This is legal, but rather pointless unless you're just writing sample code for demonstration purposes.

It's like we wrote "5" down on a slip of paper and passed it to the **`plus_one()`** function, and it went through the trouble of adding one, and writing "6" on a slip of paper and passing it back to us, and then we merely just throw it in the trash without looking at it. We're such bastards.

I have said the word "value" a number of times, and there's a good reason for that. When we pass parameters to functions, we're doing something commonly referred to as *passing by value*. This warrants its own subsection.

5.1. 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*. (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.)

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);

    return 0;
}
```

Wait a minute, wait a minute--hold it, hold it! What's this `void` return type on this function? Am I trying to pull a fast one on you? Not at all. This just means that the function doesn't return any value. Relax!

So anyway, if I might be allowed to get on with it, you might 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? None! Ha!

That's why in the previous example with the **`plus_one()`** function, we returned 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; people 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.

5.2. Function Prototypes

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

This isn't quite strictly true. You can notify the compiler in advance that you'll be using a function of a certain type that has a certain parameter list 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();

    return 0;
}

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!

6. Variables, The Sequel

Just when you thought it was safe to know everything there was to know about variables, this section of the guide lunges at you from the darkness! What?! There's more?

Yes, I'm sorry, but I'm afraid there is. We're going to talk about a couple things in this section that increase the power you have over variables *TO THE EXTREME*. Yes, by now you realize that melodrama is a well-respected part of this guide, so you probably weren't even taken off-guard by that one, ironically.

Where was I? Oh, yes; let's talk about variable *scope* and *storage classes*.

6.1. “Up Scope”

You recall how in some of those functions that we previously defined there were variables that were visible from some parts of the program but not from others? Well, if you can use a variable from a certain part of a program, it's said to be *in scope* (as opposed to *out of scope*.) A variable will be in scope if it is declared inside the block (that is, enclosed by squirrley braces) that is currently executing.

Take a look at this example:

```
int frotz(int a)
{
    int b;

    b = 10; /* in scope (from the local definition) */
    a = 20; /* in scope (from the parameter list) */
    c = 30; /* ERROR, out of scope (declared in another block, in main()) */
}

int main(void)
{
    int c;

    c = 20; /* in scope */
    b = 30; /* ERROR, out of scope (declared above in frotz()) */

    return 0;
}
```

So you can see that you have to have variables declared locally for them to be in scope. Also note that the parameter `a` is also in scope for the function **`frotz()`**

What do I mean by *local variables*, anyway? These are variable that exist and are visible only in a single basic block of code (that is, code that is surrounded by squirrley braces) and, basic blocks of code within them. For instance:

```
int main(void)
{  /* start of basic block */
    int a = 5; /* local to main() */

    if (a != 0) {
        int b = 10; /* local to if basic block */

        a = b; /* perfectly legal--both a and b are visible here */
    }

    b = 12; /* ERROR -- b is not visible out here--only in the if */

    { /* notice I started a basic block with no statement--this is legal */
```

```
        int c = 12;
        int a; /* Hey!  Wait!  There was already an "a" out in main! */

        /* the a that is local to this block hides the a from main */
        a = c; /* this modified the a local to this block to be 12 */
    }

    /* but this a back in main is still 10 (since we set it in the if): */
    printf("%d\n", a);

    return 0;
}
```

There's a lot of stuff in that example, but all of it is basically a presentation of a simple rule: when it comes to local variables, you can only use them in the basic block in which they are declared, or in basic blocks within that. Look at the “ERROR” line in the example to see exactly what *won't* work.

Let's digress for a second and take into account the special case of parameters passed to functions. These are in scope for the entire function and you are free to modify them to your heart's content. They are just like local variables to the function, except that they have copies of the data you passed in, obviously.

```
void foo(int a)
{
    int b;

    a = b; /* totally legal */
}
```

6.1.1. Global variables

There are other types of variables besides locals. There are *global variables*, for instance. Sounds grand, huh. Though they're aren't exactly the chicken's pajamas for a number of reasons, they're still a powerful piece of the language. Wield this power with care, since you can make code that's very hard to maintain if you abuse it.

A global variable is visible thoughout the entire file that it is defined in (or declared in--more on that later). So it's just like a local, except you can use it from anyplace. I guess, then, it's rather not like a local at all. But here's an example:

```
#include <stdio.h>

/* this is a global variable.  We know it's global, because it's */
/* been declared in "global scope", and not in a basic block somewhere */
int g = 10;

void afunc(int x)
{
    g = x; /* this sets the global to whatever x is */
}

int main(void)
{
    g = 10; /* global g is now 10 */
    afunc(20); /* but this function will set it to 20 */
    printf("%d\n", g); /* so this will print "20" */

    return 0;
}
```

Remember how local variables go on the stack? Well, globals go on *the heap*, another chunk of memory. And never the twain shall meet. You can think of the heap as being more "permanent" than the stack, in many ways.

Now, I mentioned that globals can be dangerous. How is that? Well, one thing you could imagine is a large-scale project in which there were a bazillion globals declared by a bazillion different programmers. What if they named them the same thing? What if you thought you were using a local, but you had forgotten to declare it, so you were using the global instead?

(Ooo. That's a good side note: if you declare a local with the same name as a global, it hides the global and all operations in the scope will take place on the local variable.)

What else can go wrong? Sometimes using global variables encourages people to not structure their code as well as they might have otherwise. It's a good idea to not use them until there simply is no other reasonable way to move data around.

Another thing to consider is this: does it actually make sense to have this data stored globally for all to see? For example, if you have a game where you use "the temperature of the world" in a lot of different places, that might be a good candidate for a global varible. Why? Because it's a pain to pass it around, and everyone has the same need for it.

On the other hand, "the temperature of this guy's little finger" might not be of so much interest to the rest of the universe. It'd be better to store that data in a way that's more associated with the guy than globally. We'll talk more later about associating data with things (nice and vague, huh?) later.

6.2. Storage Classes

What is a storage class? It's a class of storing variables.

You're welcome.

Don't get this confused with any C++ class, either, since it's not at all the same thing.

So what does a storage class declaration do? It tells the compiler where to store the data, such as on the stack or on the heap, or if variable data storage is already declared elsewhere.

"What?"

Let's just get some examples in there and it'll make more sense.

6.2.1. Gimme some static!

Ready? Here's an example: I'm sitting on BART (the Bay Area Rapid Transit subway) as I type this on my way home from work. There is a young couple happily canoodling each other in the seat in front of me, and it's completely distracting.

...Er, what. No, an example! Yes! Ok, here it is:

```
void print_plus_one(void)
{
    static int a=0; /* static storage class! */

    printf("%d\n", a);

    a++;  /* increment the static value */
}

int main(void)
{
    print_plus_one(); /* prints "0" */
    print_plus_one(); /* prints "1" */
    print_plus_one(); /* prints "2" */
    print_plus_one(); /* prints "3" */
    print_plus_one(); /* prints "4" */

    return 0;
```

```
}
```

What have we here? How can this magic be? Isn't that a local variable in **`print_plus_one()`** and doesn't it get allocated on the stack and doesn't it go away after the function returns? How can it possibly remember the value from the last call?

The answer: it uses the modern day magic that is the `static` keyword. This directive (given before the type) tells the compiler to actually store this data on the heap instead of the stack! Ooooo! Remember how the heap can be thought of as more permanent? Well, the value gets initialized once (in the definition), and it never gets initialized again, so all operations on it are cumulative.

You'll probably see more use for this later, but it's common enough that you should know about it.

6.2.2. Other Storage Classes

There are other storage classes, yes. The default is `auto`, which you never see in the wild since it's default.

Another is `extern` which tells the compiler that the definition of the variable is in a different file. This allows you to reference a global variable from a file, even if its definition is somewhere else. It would be illegal to define it twice, and since it's global it'd be nice to have it available across different source files.

I know, I know. It's hard to imagine now, but programs do get big enough to span multiple files eventually. `:-)`

7. 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 permantly in place, cursing you to a life at the keyboard.

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.

7.1. 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 of data. The number that represents each box is called its *address*.

Now, not all data types use just a byte. For instance, a `long` is often four bytes, but it really depends on the system. You can use the **`sizeof()`** operator to determine how many bytes of memory a certain type uses. (I know, **`sizeof()`** looks more like a function than an operator, but there we are.)

```
printf("a long uses %d bytes of memory\n", sizeof(long));
```

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, 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.

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 whereever. 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 hexidecimal instead of decimal), but it is merely the number of the box the data is stored in. (Or the number of the box that the first byte of data is stored in, if the data is multi-byte.) In virtually all circumstances, including this one, the actual value of the number printed is unimportant to you, and I show it here only for demonstration of the **`address-of`** operator.

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

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

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

    return 0;
}
```

On my laptop, this prints:

```
The value of i is 10, and its address is 0xbffff964
```

(I can show off a bit here, and say that from experience the address to me looks like its on the stack...and it is, since it's a local variable, and all locals go on the stack. Am I cool or what. Don't answer that.)

7.2. 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.`

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" */

    return 0;
}
```

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 */
```

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.

7.3. Dereferencing

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 interchangably.

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 (AKA "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.

Here's a full-blown example:

```
#include <stdio.h>

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

    p = &i; // p now points to i

    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!

    return 0;
}
```

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`.

7.4. 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? 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 p */
}

int main(void)
{
    int i = 10;

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

    increment(&i); /* note the address-of; turns it into a pointer */

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

    return 0;
}
```

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 *`&i`* (in **`main()`**) and the copy of the pointer *`p`* (in **`increment()`**) point to the same address. 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!

8. Structures

You've been messing with variables for quite some time now, and you've made a bunch of them here and there to do stuff, yes? Usually you bundle them into the same basic block and let it go at that.

Sometimes, though, it makes more sense to put them together into a *structure*. This is a construct that allows you to logically (or even illogically, if the fancy takes you) group variables into, uh, groups. You can then reference the group as a whole. One place this comes in really handily is if you want to pass 13 bazillion parameters to a function, but don't want the function declaration to be that long. You just put all the variables into a structure (or `struct`, as it is normally called), and then pass that structure to the function. (Actually, people usually pass a pointer to the structure, but we'll get to that later.)

So how do you use one of these things? First off, the `struct` itself is a new type. If you make a variable that is a `struct foo`, it's type is "`struct foo`", just as the number 12 is of type "`int`". This is a little bit spooky because this could be the first time you've created a new type, and it might be unclear how that works.

Adding to the confusion, it turns out there are multiple ways to create and use a new `struct` type, and barely any of them are particularly intuitive. We'll have an example of one way to declare a new `struct` here:

```
#include <stdio.h>

/* Here we declare the type so we can use it later: */
struct stuff {
    int val;
    float b;
};

/* Note that we don't actually have any variables of that type, yet. */

int main(void)
{
    /* ok, now let's declare a variable "s" of type "struct stuff" */
    struct stuff s;


    s.val = 3490;  /* assignment into a struct! */
    s.b = 3.14159;

    printf("The val field in s is: %d\n", s.val);

    return 0;
}
```

The compiler allows us to predeclare a `struct` like in the example. We can then use it later, like we do in **`main()`** to assign values into it. When we say things like `s.val = 3490`, we are using a special operator to access the `val` field, known as the *dot operator* (`.`).

8.1. Pointers to `structs`

Now let's talk a bit about how to pass these `structs` around to other functions and stuff. I mentioned before that you probably want to pass a pointer to the `struct` instead of the `struct` itself. Why?

Don't you hate it when the professor asks you a question like that, but you're too tired in lecture to care to begin to think about it? “Just tell me and we'll get on with it,” is what you're thinking, isn't it.

Fine! Be that way! Well, remember that when you pass parameters to functions, and I'll clear my throat here in preparation to say again, *EVERY PARAMETER WITHOUT FAIL GETS COPIED ONTO THE STACK when you call a function!* So if you have a huge `struct` that's like 80,000 bytes in size, it's going to copy that onto the stack when you pass it. That takes time.

Instead, why not pass a pointer to the `struct`? I hear you--doesn't the pointer have to get copied on the stack then? Sure does, but a pointer is, these days, only 4 or 8 bytes, so it's much easier on the machine, and works faster.

And there's even a little bit of *syntactic sugar* to help access the fields in a pointer to a `struct`. For those that aren't aware, syntactic sugar is a feature of a compiler that simplifies the code even though there is another way to accomplish the same thing. For instance, I've already mentioned the **+=** a number of times...what does it do? Did I ever tell you? To be honest, in all the excitement, I've forgotten myself. Here is an example that shows how it works like another two operators, but is a little bit easier to use. It is syntactic sugar:

```
i = i + 12;  /* add 12 to i */
i += 12;     /* <-- does exactly the same thing as "i = i + 12" */
```

You see? It's not a necessary operator because there's another way to do the same thing, but people like the shortcut.

But we are way off course, buster. I was talking about pointers to `struct`s, and here we are talking about how to access them. Here's an example wherein we have a `struct` variable, and another variable that is a pointer to that `struct` type, and some usage for both (this will use `struct stuff` from above):

```
#include <stdio.h>

/* Here we declare the type so we can use it later: */
struct antelope {
    int val;
    float something;
};

int main(void)
{
    struct antelope a;
    struct antelope *b; /* this is a pointer to a struct antelope */

    b = &a; /* let's point b at a for laughs and giggles */

    a.val = 3490; /* normal struct usage, as we've already seen */

    /* since b is a pointer, we have to dereference it before we can */
    /* use it:                                                       */

    (*b).val = 3491;

    /* but that looks kinda bad, so let's do the exact same thing  */
    /* except this time we'll use the "arrow operator", which is a */
    /* bit of syntactic sugar:                                     */

    b->val = 3491; /* EXACTLY the same as (*b).val = 3491; */

    return 0;
}
```

So here we've seen a couple things. For one, we have the manner with which we dereference a pointer to a `struct` and then use the dot operator (`.`) to access it. This is kind of the classic way of doing things: we have a pointer, so we dereference it to get at the variable it points to, then we can treat it as if it *is* that variable.

But, syntactic sugar to the rescue, we can use the *arrow operator* (**->**) instead of the dot operator! Saves us from looking ugly in the code, doesn't? The arrow operator has a build-in deference, so you don't have to mess with that syntax when you have a pointer to a `struct`. So the rule is this: if you have a `struct`, use the dot operator; if you have a *pointer* to a `struct`, use the arrow operator (**->**).

8.2. Passing `struct` pointers to functions

This is going to be a very short section about passing pointers to structs to functions. I mean, you already know how to pass variables as paramters to functions, and you already know how to make and use pointers to `struct`s, so there's really not much to be said here. An example should show it straightforwardly enough. ("Straightforwardly"--there's an oxymoronic word if ever I saw one.)

Grrrr. Why do I always sit on this side of the bus? I mean, I know that the sun streams in here, and I can barely read my screen with the bright light blazing away against it. You think I'd learn by now. So *anyway*...

```
#include <stdio.h>

struct mutantfrog {
    int num_legs;
    int num_eyes;
};

void build_beejs_frog(struct mutantfrog *f)
{
    f->num_legs = 10;
    f->num_eyes = 1;
}

int main(void)
{
    struct mutantfrog rudolph;

    build_beejs_frog(&rudolph);  /* passing a pointer to the struct */

    printf("leg count: %d\n", rudolph.num_legs); /* prints "10" */
    printf("eye count: %d\n", rudolph.num_eyes); /* prints "1" */

    return 0;
}
```

Another thing to notice here: if we passed the `stuct` instead of a pointer to the `struct`, what would happen in the function **`build_beejs_frog()`** when we changed the values? That's right: they'd only be changed in the local copy, and not back at out in **`main()`**. So, in short, pointers to `struct`s are the way to go when it comes to passing `struct`s to functions.

Just a quick note here to get you thinking about how you're going to be breaking stuff down into basic blocks. You now have a lot of different tools at your disposal: loops, conditionals, `struct`s, and especially functions. Remember back on how you learned to break down projects into little pieces and see what you'd use these individual tools for.

9. Arrays

What a wide array of information we have for you in this section. **BLAM*!* We're sorry--that pun has been taken out and shot.

An array: a linear collection of related data. Eh? It's a continuous chunk of memory that holds a number of identical data types. Why would we want to do that? Well, here's an example without arrays:

```
int age0;
int age1;
int age2;
int age3;
int age4;

age1 = 10;
printf("age 1 is %d\n", age1);
```

And here is that same example using the magical power of arrays:

```
int age[5];

age[1] = 10;
printf("age 1 is %d\n", age[1]);
```

Ooooo! See how much prettier that is? You use the square-brackets notation (`[]`) to access *elements* in the array by putting an `int` variable in there or, like we did in the example, a constant number. Since you can *index* the array using a variable, it means you can do things in loops and stuff. Here's a better example:

```
int i;
int age[5] = {10,20,25,8,2};

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

whoa--we've done a couple new things here. for one, you'll notice in the array definition that we've initialized the entire array at once. this is allowed *in the definition only*. once you get down to the code, it's too late to initialize the array like this, and it must be done an element at a time.

the other thing to notice is how we've accessed the array element inside the for loop: using *`age[i]`*. so what happens in this for is that *`i`* runs from 0 to 4, and so each *`age`* is printed out in turn, like so:

```
age 0 is 10
age 1 is 20
age 2 is 25
age 3 is 8
age 4 is 2
```

why does it run from 0 to 4 instead of 1 to 5? good question. the easiest answer is that the array index numbers are *zero-based* instead of one-based. that's not really an answer, i know...well, it turns out that most (if not all) processors use zero-based indexing in their machine code for doing memory access, so this maps very well to that.

You can make arrays of any type you desire, including `struct`s, and pointers.

9.1. Passing arrays to functions

it's pretty effortless to pass an array to a function, but things get a little wierd. the weirdest part is that when you pass the “whole” array by putting its name in as a parameter to the function, it's actually a *pointer to the first element of the array* that gets passed in for you.

now, inside the function, you can still access the array using array notation (`[]`). here's an example:

```
void init_array(int a[], int count)
{
    int i;

    /* for each element, set it equal to its index number times 10: */

    for(i = 0; i < count; i++)
        a[i] = i * 10;
}

int main(void)
{
    int mydata[10];

    init_array(my_data, 10);  /* note lack of [] notation */

    return 0;
}
```

A couple things to note here are that we didn't have to specify the array *dimension* (that is, how many elements are in the array) in the declaration for **`init_array()`**. We could have, but we didn't have to.

Also, since an array in C has no built-in concept of how big it is (i.e. how many elements it holds), we have to be nice and cooperative and actually pass in the array size separately into the function. We use it later in the for to know how many elements to initialize.

Hey! We didn't use the address-of operator in the call! Wouldn't that make a copy of the array onto the stack? Isn't that bad? Well, no.

When you have an array, leaving off the and square brackets gives you a *pointer* to the first element of the array. (You can use the address-of operator if you want, but it actually results in a different pointer type, so it might not be what you expect.) So it is, in fact, a pointer to the array that's getting copied onto the stack for the function call, not the entire array.

Right about now, you should be recognizing that you can use arrays to hold a lot of things, and that could serve you quite well in your projects which often involve collections of data. For instance, let's say:

We have a virtual world where we have a number of virtual creatures that run around doing virtual things. Each creature has a real X and Y coordinate. There are 12 creatures. Each step of the simulation, the creatures will process their behavior algorithm and move to a new position. The new position should be displayed.

Uh oh! Time for building blocks! What do we have? Ok, we need an X and Y coordinate for the creatures. We need 12 creatures. We need a construct that repeatedly processes the behavior for each. And we need output.

So the coordinates of the creature. There are a few ways to do this. You could have two arrays of 12 elements, one to hold the X and one to hold the Y coordinate. (These are known as *parallel arrays*.) But let's instead try to think of a way that we could bundle those data together. I mean, they're both related to the same virtual creature, so wouldn't it be nice to have them somehow logically related? If only there were a way...but wait! That's right! A `struct`!

```
struct creature {
    float x;
    float y;
};
```

There we go. Now we need 12 of them; 12 items of type `struct creature`. What's a good way of holding a collection of identical types? Yes, yes! Array!

```
    struct creature guys[12];
```

So what else--we need to be able to repeatedly execute the behavior of these creatures. Like looping over and over and over...yes, a loop would be good for that. But how many times should we loop? I mean, the specification didn't tell us that, so it's an excellent question. Often we'll loop until some exit condition is true (like the user presses ESCAPE or something like that), but since the spec writer didn't say, let's just loop forever.

```
    for(;;) {  /* loop forever */
```

Now what? We need to write the behavior of the creatures and put it in the loop, right? Since this is a lot of self-contained code we're about to write, we might as well stick it in a function and call that function for each of the creatures. But I'll leave the actual implementation of that function up to you, the reader. `:-)`

```
        for(i = 0; i < 12; i++) {
            execute_behavior(&(guy[i]));
        }
```

You notice that we did use the address-of operator there in the call. In this case, we're not passing the whole array; we're just passing a pointer to a single element in the array. It's not always necessary to do that (you could have it copy the single element in the call), but since this is a `struct`, I pass a pointer to keep the memory overhead low.

The last thing to do was to output the information. How this is done should be in the spec, but isn't. It would be cool to do a hi-resolution screen with little icons for where the creatures are, but that is currently beyond the scope of what we're doing here, so we'll just write a simple thing to print out the creatures.

One final note--it's always a good idea to initialize the data before using it, right? So I'll write a function that initializes the creatures, too, before we use it. How it initializes them is also undefined in the spec, so I'll arbitrarily set them up in a diagonal line.

Completed (except for the behavior) code:

```
#include <stdio.h>

struct creature {
    float x;
    float y;
};

void execute_behavior(struct creature *c)
{
    /* does nothing now */
    /* --you'll have to code it if you want them to ever move! */
}

main(int main(void)
{
    int i;
    struct creature guys[12];

```

```
    /* initialize them--could be its own function: */

    for(i = 0; i < 12; i++) {
        guys[i].x = (float)i; /* (float) is a "cast"--it changes the type! */
        guys[i].y = (float)i;
    }

    /* main loop */

    for(;;) {   /* loop forever */

        /* have them do their thing: */

        for(i = 0; i < 12; i++) {
            execute_behavior(&(guy[i]));
        }

        /* output the result */
        for(i = 0; i < 12; i++) {
            printf("creature %d: (%.2f, %.2f)\n", i, guys[i].x, guys[i].y);
        }
    }

    return 0;
}
```

I threw in a *cast* there in the code: `(float)`. See *`i`* is an `int`, but each of the fields *`guys[i].x`* and *`guys[i].y`* is a `float`. The cast changes the expression right after it, in this case "*`i`*" to the specified type. It's always a good idea to have the same types on both sides of the assignment operator (**`=`**).

Another new thing is the "`%.2f`" in the **`printf()`** format string. A plain "`%f`" means to print a `float`, which is what we're passing it. The addtional "`.2`" means print it using two decimal places. You can leave the "`.2`" off and see what happens. `:-)`

10. Strings

A string is a theoretical construct in modern physics that is used to help explain the very fabric of the universe itself.

This is *exactly* the same as a C string, except that it is, in fact, completely different.

A string in C is a sequence of bytes in memory that usually contains a bunch of letters. Constant strings in C are surrounded by double quotes (`"`). You may have seen strings before in such programming blockbusters, such as Hello World:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");

    return 0;
}
```

You've spotted the string `"Hello, World!\n"` in there, haven't you.

What type is that string, anyway? Well, it turns out that a constant string (that is, one in double quotes) is of type `char*`. But you can also put a string in a `char` array, if you so desire. The `char*` points at the first character in the string. Examples:

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

printf("%s\n", s);    /* prints "Hello!" */
printf("%c\n", *s);   /* prints 'H' */
printf("%c\n", s[0]); /* prints 'H' */
printf("%c\n", s[1]); /* prints 'e' */
printf("%c\n", s[4]); /* prints 'o' */
```

(Note the two new format specifiers for **`printf()`** here: `%c` for printing a single `char`, and `%s` for printing a string! Ain't that exciting!)

And look here, we're accessing this string in a whole variety of different ways. We're printing the whole thing, and we're printing single characters.

You can also initialize `char` arrays during their definition, just like other arrays:

```
char s[20] = "The aliens are coming!";
```

And you can change the array elements on the fly, too by simply assigning into it:

```
char s[20] = "Give me $10!";

printf("%s\n", s);  /* prints "Give me $10!" */

s[9] = '8';
printf("%s\n", s);  /* prints "Give me $80!" */
```

In this case, note that we've put a constant `char` on the right side of the assignment. Constant `char`s use the single quote (`'`).

One more thing to remember is that when you specify an array by name without the square brackets, it's just like using a pointer to the beginning of the array. So you can actually do assignments like this:

```
char a[20] = "Cats are better.";
char *p;
```

```
p = a;  /* p now points to the first char in array a */
```

One more thing: strings in C end with a NUL character. That is, a zero. It is most properly written as '\0', You can truncate a string to zero length by assigning '\0' to the first byte in the string. Also, all the standard string functions assume the string ends with a NUL character.

Standard string functions, did I say? Yes, I did.

C provides a whole metric slew of functions that you can use to modify strings, stick them together, parse them apart, and so on. Check out the reference section for all the brilliant *string processing power* at your disposal. It is your responsibility as a citizen of the Planet Earth to wield this ultimate power wisely! (And pay me $80.)

11. Dynamic Memory

Well, aren't you looking dynamic this morning, reader? Or is it evening? I always lose track of these things.

Up until now, we've been talking about memory that pretty much is set up at the beginning of the program run. You have constant strings here and there, arrays of predeclared length, and variables all declared ahead of time. But what if you have something like the following?

Assignment: Implement a program that will read an arbitrary number of integers from the keyboard. The first line the user enters will be the number of `int`s to read. The `int`s themselves will appear on subsequent lines, one `int` per line.

Yes, it's that time again: break it up into component parts that you can implement. You'll need to read lines of text from the keyboard (there's a cool little function called **`fgets()`** that can help here), and the first line you'll need to convert to an integer so you know how many more lines to read. (You can use **`atoi()`**, read “ascii-to-integer” to do this conversion.) Then you'll need to read that many more strings and store them...where?

Here's where dynamic memory can help out--we need to store a bunch of `int`s, but we don't know how many until after the program has already started running. What we do is find out how many `int`s we need, then we calculate how many bytes we need for each, multiply those two numbers to get the total number of bytes we need to store everything, and then ask the OS to allocate that many bytes for us on the heap for us to use in the manner we choose. In this case, we're choosing to store `int`s in there.

There are three functions we're going to talk about here. Well, make that four functions, but one is just a variant of another: **`malloc()`** (allocate some memory for us to use), **`free()`** (release some memory that **`malloc()`** gave us earlier), **`realloc()`** (change the size of some previously allocated memory), and **`calloc()`** (just like **`malloc()`**, except clears the memory to zero.)

Using these functions in unison results in a beautifully intricate dance of data, ebbing and flowing with the strong tidal pull of the dedicated user's will.

Yeah. Let's cut the noise and get on with it here.

11.1. `malloc()`

This is the big one: he's the guy that gives you memory when you ask for it. It returns to you a pointer to a chunk of memory of a specified number of bytes, or `NULL` if there is some kind of error (like you're out of memory). The return type is actually `void*`, so it can turn into a pointer to whatever you want.

Since **`malloc()`** operates in bytes of memory and you often operate with other data types (e.g. “Allocate for me 12 `int`s.”), people often use the **`sizeof()`** operator to determine how many bytes to allocate, for example:

```
int *p;

p = malloc(sizeof(int) * 12); // allocate for me 12 ints!
```

Oh, and that was pretty much an example of how to use **`malloc()`**, too. You can reference the result using pointer arithmetic or array notation; either is fine since it's a pointer. But you should really check the result for errors:

```
int *p;

p = malloc(sizeof(float) * 3490); // allocate 3490 floats!
```

```
if (p == NULL) {
    printf("Horsefeathers!  We're probably out of memory!\n");
    exit(1);
}
```

More commonly, people pack this onto one line:

```
if ((p = malloc(100)) == NULL) { // allocate 100 bytes
    printf("Ooooo!  Out of memory error!\n");
    exit(1);
}
```

Now remember this: you're allocating memory on the heap and there are only two ways to ever get that memory back: 1) your program can exit, or 2) you can call **`free()`** to free a **`malloc()`**'d chunk. If your program runs for a long time and keeps **`malloc()`**ing and never **`free()`**ing when it should, it's said to "leak" memory. This often manifests itself in a way such as, "Hey, Bob. I started your print job monitor program a week ago, and now it's using 13 terabytes of RAM. Why is that?"

Be sure to avoid memory leaks! **`free()`** that memory when you're done with it!

11.2. `free()`

Speaking of how to free memory that you've allocated, you do it with the implausibly-named **`free()`** function.

This function takes as its argument a pointer that you've picked up using **`malloc()`** (or **`calloc()`**). And it releases the memory associated with that data. You really should never use memory after it has been **`free()`**'d. It would be Bad.

So how about an example:

```
int *p;

p = malloc(sizeof(int) * 37); // 37 ints!

free(p);  // on second thought, never mind!
```

Of course, between the **`malloc()`** and the **`free()`**, you can do anything with the memory your twisted little heart desires.

11.3. `realloc()`

`realloc()` is a fun little function that takes a chunk of memory you allocated with **`malloc()`** (or **`calloc()`**) and changes the size of the memory chunk. Maybe you thought you only needed 100 `int`s at first, but now you need 200. You can **`realloc()`** the block to give you the space you need.

This is all well and good, except that **`realloc()`** might have to *move your data* to another place in memory if it can't, for whatever reason, increase the size of the current block. It's not omnipotent, after all.

What does this mean for you, the mortal? Well in short, it means you should use **`realloc()`** sparingly since it could be an expensive operation. Usually the procedure is to keep track of how much room you have in the memory block, and then add another big chunk to it if you run out. So first you allocate what you'd guess is enough room to hold all the data you'd require, and then if you happened to run out, you'd reallocate the block with the next best guess of what you'd require in the future. What makes a good guess depends on the program. Here's an example that just allocates more "buckets" of space as needed:

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

```
#define INITIAL_SIZE 10
#define BUCKET_SIZE 5

static int data_count; // how many ints we have stored
static int data_size;  // how many ints we *can* store in this block
static int *data;      // the block of data, itself

int main(void)
{
    void add_data(int new_data); // function prototype
    int i;

    // first, initialize the data area:
    data_count = 0;
    data_size = INITIAL_SIZE;
    data = malloc(data_size * sizeof(int)); // allocate initial area

    // now add a bunch of data
    for(i = 0; i < 23; i++) {
        add_data(i);
    }

    return 0;
}

void add_data(int new_data)
{
    // if data_count == data_size, the area is full and
    // needs to be realloc()'d before we can add another:

    if (data_count == data_size) {
        // we're full up, so add a bucket
        data_size += BUCKET_SIZE;
        data = realloc(data, data_size * sizeof(int));
    }

    // now store the data
    *(data+data_count) = new_data;

    // ^^^ the above line could have used array notation, like so:
    //  data[data_count] = new_data;

    data_count++;
}
```

In the above code, you can see that a potentially expensive **`realloc()`** is only done after the first 10 `int`s have been stored, and then again only after each block of five after that. This beats doing a **`realloc()`** every time you add a number, hands down.

(Yes, yes, in that completely contrived example, since I know I'm adding 23 numbers right off the get-go, it would make much more sense to set *`INITIAL_SIZE`* to 25 or something, but that defeats the whole purpose of the example, now, doesn't it?)

11.4. `calloc()`

Since you've already read the section on **`malloc()`** (you have, right?), this part will be easy! Yay! Here's the scoop: **`calloc()`** is just like **`malloc()`**, except that it 1) clears the memory to zero for you, and 2) it takes two parameters instead of one.

The two parameters are the number of elements that are to be in the memory block, and the size of each element. Yes, this is exactly like we did in **`malloc()`**, except that **`calloc()`** is doing the multiply for you:

```
// this:
p = malloc(10 * sizeof(int));

// is just like this:
p = calloc(10, sizeof(int));
// (and the memory is cleared to zero when using calloc())
```

The pointer returned by **`calloc()`** can be used with **`realloc()`** and **`free()`** just as if you had used **`malloc()`**.

The drawback to using **`calloc()`** is that it takes time to clear memory, and in most cases, you don't need it clear since you'll just be writing over it anyway. But if you ever find yourself **`malloc()`**ing a block and then setting the memory to zero right after, you can use **`calloc()`** to do that in one call.

I wish this section on **`calloc()`** were more exciting, with plot, passion, and violence, like any good Hollywood picture, but...this is C programming we're talking about. And that should be exciting in its own right. Sorry!

12. More Stuff!

This is the section where we flesh out a bunch of the stuff we'd done before, except in more detail. We even throw a couple new things in there for good measure. You can read these sections in any order you want and as you feel you need to.

12.1. Pointer Arithmetic

Pointer *what*? Yeah, that's right: you can perform math on pointers. What does it mean to do that, though? Well, pay attention, because people use *pointer arithmetic* all the time to manipulate pointers and move around through memory.

You can add to and subtract from pointers. If you have a pointer to a `char`, incrementing that pointer moves to the next `char` in memory (one byte up). If you have a pointer to an `int`, incrementing that pointer moves to the next `int` in memory (which might be four bytes up, or some other number depending on your CPU architecture.) It's important to know that the number of bytes of memory it moves differs depending on the type of pointer, but that's actually all taken care of for you.

```
/* This code prints: */
/*  50               */
/*  99               */
/*  3490             */

int main(void)
{
    int a[4] = { 50, 99, 3490, 0 };
    int *p;

    p = a;
    while(*p > 0) {
        printf("%i\n", *p);
        p++; /* go to the next int in memory */
    }

    return 0;
}
```

What have we done! How does this print out the values in the array? First of all, we point *p* at the first element of the array. Then we're going to loop until what *p* points at is less than or equal to zero. Then, inside the loop, we print what *p* is pointing at. Finally, *and here's the tricky part*, we *increment the pointer*. This causes the pointer to move to the next `int` in memory so we can print it.

In this case, I've arbitrarily decided (yeah, it's shockingly true: I just make all this stuff up) to mark the end of the array with a zero value so I know when to stop printing. This is known as a *sentinel value*...that is, something that lets you know when some data ends. If this sounds familiar, it's because you just saw it in the section on strings. Remember--strings end in a zero character (`'\0'`) and the string functions use this as a sentinel value to know where the string ends.

Lots of times, you see a for loop used to go through pointer stuff. For instance, here's some code that copies a string:

```
char *source = "Copy me!";
char dest[20];  /* we'll copy that string into here */

char *sp;  /* source pointer */
```

```
char *dp;  /* destination pointer */

for(sp = source, dp = dest; *sp != '\0'; sp++, dp++) {
    *dp = *sp;
}

printf("%s\n", dest); /* prints "Copy me!" */
```

Looks complicated! Something new here is the *comma operator* (`,`). The comma operator allows you to stick expressions together. The total value of the expression is the rightmost expression after the comma, but all parts of the expression are evaluated, left to right.

So let's take apart this for loop and see what's up. In the initialization section, we point *`sp`* and *`dp`* to the source string and the destination area we're going to copy it to.

In the body of the loop, the actual copy takes place. We copy, using the assignment operator, the character that the source pointer points to, to the address that the destination pointer points to. So in this way, we're going to copy the string a letter at a time.

The middle part of the for loop is the continuation condition--we check here to see if the source pointer points at a NUL character which we know exists at the end of the source string. Of course, at first, it's pointing at `'C'` (of the “Copy me!” string), so we're free to continue.

At the end of the for statement we'll increment both *`sp`* and *`dp`* to move to the next character to copy. Copy, copy, copy!

12.2. typedef

This one isn't too difficult to wrap your head around, but there are some strange nuances to it that you might see out in the wild. Basically typedef allows you to make up an alias for a certain type, so you can reference it by that name instead.

Why would you want to do that? The most common reason is that the other name is a little bit too unwieldy and you want something more concise...and this most commonly occurs when you have a `struct` that you want to use.

```
struct a_structure_with_a_large_name {
    int a;
    float b;
};

typedef struct a_structure_with_a_large_name NAMESTRUCT;

int main(void)
{
    /* we can make a variable of the structure like this: */
    struct a_structure_with_a_large_name one_variable;

    /* OR, we can do it like this: */
    NAMESTRUCT another_variable;

    return 0;
}
```

In the above code, we've defined a type, `NAMESTRUCT`, that can be used in place of the other type, `struct a_structure_with_a_large_name`. Note that this is now a full-blown type; you can use it in function calls, or whereever you'd use a “normal” type. (Don't tell typedef'd types they're not normal--it's impolite.)

You're probably also wondering why the new type name is in all caps. Historically, typedef'd types have been all caps in C by convention (it's certainly not necessary.) In C++, this is no longer the case and people use mixed case more often. Since this is a C guide, we'll stick to the old ways.

(One thing you might commonly see is a `struct` with an underscore before the `struct` tag name in the typedef. Though technically illegal, many programmers like to use the same name for the `struct` as they do for the new type, and putting the underscore there differentiates the two visaully. But you shouldn't do it.)

You can also typedef "anonymous" `struct`s, like this

```
typedef struct {
    int a;
    float b;
} someData;
```

So then you can define variables as type `someData`. Very exciting.

12.3. enum

Sometimes you have a list of numbers that you want to use to represent different things, but it's easier for the programmer to represent those things by name instead of number. You can use an `enum` to make symbolic names for integer numbers that programmers can use later in their code in place of `int`s.

(I should note that C is more relaxed that C++ is here about interchanging `int`s and `enum`s. We'll be all happy and C-like here, though.)

Note that an `enum` is a type, too. You can typedef it, you can pass them into functions, and so on, again, just like "normal" types.

Here are some enums and their usage. Remember--treat them just like `int`s, more or less.

```
enum fishtypes {
    HALIBUT,
    TUBESNOUT,
    SEABASS,
    ROCKFISH
};

int main(void)
{
    enum fishtypes fish1 = SEABASS;
    enum fishtypes fish2;

    if (fish1 == SEABASS) {
        fish2 = TUBESNOUT;
    }

    return 0;
}
```

Nothing to it--they're just symbolic names for unique numbers. Basically it's easier for other programmers to read and maintain.

Now, you can print them out using `%d` in **`printf()`**, if you want. For the most part, though, there's no reason to know what the actual number is; usually you just want the symbolic representation.

But, since I know you're dying of curiosity, I might as well tell you that the `enum`s start at zero by default, and increase from there. So in the above example, `HALIBUT` would be 0, `TUBESNOUT` would be 1, and `ROCKFISH` would be 3.

If you want, though, you can override any or all of these:

```
enum frogtypes {
    THREELEGGED=3,
    FOUREYED,
    SIXHEADED=6
```

```
};
```

In the above case, two of the enums are explicitly defined. For `FOUREYED` (which isn't defined), it just increments one from the last defined value, so its value is 4. You can, if you're curious, have duplicate values, but why would you want to, sicko?

12.4. More `struct` declarations

Remember how, many moons ago, I mentioned that there were a number of ways to declare `struct`s and not all of them made a whole lot of sense. We've already seen how to declare a `struct` globally to use later, as well as one in a typedef situation, *comme ca*:

```
/* standalone: */

struct antelope {
    int legcount;
    float angryfactor;
};

/* or with typedef: */

typedef struct _goatcheese {
    char victim_name[40];
    float cheesecount;
} GOATCHEESE;
```

But you can also declare variables along with the `struct` declaration by putting them directly afterward:

```
struct breadtopping {
    enum toppingtype type; /* BUTTER, MARGARINE or MARMITE */
    float amount;
} mytopping;

/* just like if you'd later declared: */

struct breadtopping mytopping;
```

So there we've kinda stuck the variable defintion on the tail end of the `struct` definition. Pretty sneaky, but you see that happen from time to time in that so-called *Real Life* thing that I hear so much about.

And, just when you thought you had it all, you can actually omit the struct name in many cases. For example:

```
typedef struct { /* <--Hey!  We left the name off! */
    char name[100];
    int num_movies;
} ACTOR_PRESIDENT;
```

It's more *right* to name all your `struct`s, even if you don't use the proper name and only use the typedef'd name, but you still see those naked `struct`s here and there.

12.5. Command Line Arguments

I've been lying to you this whole time, I must admit. I thought I could hide it from you and not get caught, but you realized that something was wrong...why doesn't the **`main()`** have a return type or argument list?

Well, back in the depths of time, for some reason, !!!TODO research!!! it was perfectly acceptable to do that. And it persists to this day. Feel free to do that, in fact, But that's not telling you the whole story, and it's time you knew the *whole truth*!

Welcome to the real world:

```
int main(int argc, char **argv)
```

Whoa! What is all that stuff? Before I tell you, though, you have to realize that programs, when executed from the command line, accept arguments from the command line program, and return a result to the command line program. Using many Unix shells, you can get the return value of the program in the shell variable *`$?`*. (This doesn't even work in the windows command shell--use !!!TODO look up windows return variable!!! instead.) And you specify parameters to the program on the command line after the program name. So if you have a program called "makemoney", you can run it with parameters, and then check the return value, like this:

```
$ makemoney fast alot
$ echo $?
2
```

In this case, we've passed two command line arguments, "fast" and "alot", and gotten a return value back in the variable *`$?`*, which we've printed using the Unix **echo** command. How does the program read those arguments, and return that value?

Let's do the easy part first: the return value. You've noticed that the above prototype for **`main()`** returns an `int`. Swell! So all you have to do is either return that value from **`main()`** somewhere, or, alternatively, you can call the function **`exit()`** with an exit value as the parameter:

```
int main(void)
{
    int a = 12;

    if (a == 2) {
        exit(3); /* just like running (from main()) "return 3;" */
    }

    return 2; /* just like calling exit(2); */
}
```

For historical reasons, an exit status of `0` has meant success, while nonzero means failure. Other programs can check your program's exit status and react accordingly.

Ok that's the return status. What about those arguments? Well, that whole definition of *`argv`* looks too intimidating to start with. What about this *`argc`* instead? It's just an `int`, and an easy one at that. It contains the total count of arguments on the command line, *including the name of the program itself*. For example:

```
$ makemoney fast alot        # <-- argc == 3
$ makemoney                  # <-- argc == 1
$ makemoney 1 2 3 4 5        # <-- argc == 6
```

(The dollar sign, above, is a common Unix command shell prompt. And that hash mark (`#`) is the command shell comment character in Unix. I'm a Unix-dork, so you'll have to deal. If you have a problem, talk to those friendly Stormtroopers over there.)

Good, good. Not much to that *`argc business`*, either. Now the biggie: *`argv`*. As you might have guessed, this is where the arguments themselves are stored. But what about that `char**` type? What do you do with that? Fortunately, you can often use array notation in the place of a dereference, since you'll recall arrays and pointers are related beasties. In this case, you can think of *`argv`* as an array of pointers to strings, where each string pointed to is one of the command line arguments:

```
$ makemoney   somewhere  somehow
$ # argv[0]    argv[1]   argv[2]   (and argc is 3)
```

Each of these array elements, *argv[0]*, *argv[1]*, and so on, is a string. (Remember a string is just a pointer to a `char` or an array of `char`s, the name of which is a pointer to the first element of the array.)

I haven't told you much of what you can do with strings yet, but check out the reference section for more information. What you do know is how to **`printf()`** a string using the "`%s`" format specifier, and you do know how to do a loop. So let's write a program that simply prints out its command line arguments, and then sets an exit status value of 4:

```
/* showargs.c */

#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    printf("There are %d things on the command line\n", argc);
    printf("The program name is \"%s\"\n", argv[0];

    printf("The arguments are:\n");

    for(i = 1; i < argc; i++) {
        printf("    %s\n", argv[i]);
    }

    return 4; /* exit status of 4 */
}
```

Note that we started printing arguments at index 1, since we already printed *argv[0]* before that. So sample runs and output (assuming we compiled this into a program called **showargs**):

```
$ showargs alpha bravo
There are 3 things on the command line
The program name is "showargs"
The arguments are:
    alpha
    bravo

$ showargs
There are 1 things on the command line
The program name is "showargs"
The arguments are:

$ showargs 12
There are 2 things on the command line
The program name is "showargs"
The arguments are:
    12
```

(The actual thing in *argv[0]* might differ from system to system. Sometimes it'll contain some path information or other stuff.)

So that's the secret for getting stuff into your program from the command line!

12.6. Multidimensional Arrays

Welcome to...the *Nth Dimension!* Bet you never thought you'd see that. Well, here we are. Yup. The Nth Dimension.

Ok, then. Well, you've seen how you can arrange sequences of data in memory using an array. It looks something like this:

!!!TODO image of 1d array

Now, imagine, if you will, a *grid* of elements instead of just a single row of them:

This is an example of a *two-dimensional* array, and can be indexed by giving a row number and a column number as the index, like this: `a[2][10]`. You can have as many dimensions in an array that you want, but I'm not going to draw them because 2D is already past the limit of my artistic skills.

So check this code out--it makes up a two-dimensional array, initializes it in the definition (see how we nest the squirrely braces there during the init), and then uses a *nested loop* (that is, a loop inside another loop) to go through all the elements and pretty-print them to the screen.

```
#include <stdio.h>

int main(void)
{
    int a[2][5] = { { 10, 20, 30, 40, 55 },  /* [2][5] == [rows][cols] */
                    { 10, 18, 21, 30, 44 } };

    int i, j;

    for(i = 0; i < 2; i++) {     /* for all the rows... */
        for(j = 0; j < 5; j++) { /* print all the columns! */
            printf("%d ", a[i][j]);
        }

        /* at the end of the row, print a newline for the next row */
        printf("\n");
    }

    return 0;
}
```

As you might well imagine, since there really is no surprise ending for a program so simple as this one, the output will look something like this:

```
10 20 30 40 55
10 18 21 30 44
```

Hold on for a second, now, since we're going to take this concept for a spin and learn a little bit more about how arrays are stored in memory, and some of tricks you can use to access them. First of all, you need to know that in the previous example, even though the array has two rows and is multidimensional, the data is stored sequentially in memory in this order: 10, 20, 30, 40, 55, 10, 18, 21, 30, 44.

See how that works? The compiler just puts one row after the next and so on.

But hey! Isn't that just like a one-dimensional array, then? Yes, for the most part, it technically is! A lot of programmers don't even bother with multidimensional arrays at all, and just use single dimensional, doing the math by hand to find a particular row and column. You can't technically just switch dimensions whenever you feel like it, Buccaroo Bonzai, because the types are different. And it'd be bad form, besides.

For instance...nevermind the “for instance”. Let's do the same example again using a single dimensional array:

```
#include <stdio.h>

int main(void)
{
    int a[10] = {   10, 20, 30, 40, 55,  /* 10 elements (2x5) */
                    10, 18, 21, 30, 44 };

    int i, j;

    for(i = 0; i < 2; i++) {     /* for all the rows... */
        for(j = 0; j < 5; j++) { /* print all the columns! */
            int index = i*5 + j; /* calc the index */
            printf("%d ", a[index]);
        }

        /* at the end of the row, print a newline for the next row */
        printf("\n");
    }

    return 0;
}
```

So in the middle of the loop we've declared a local variable *index* (yes, you can do that--remember local variables are local to their block (that is, local to their surrounding squirrley braces)) and we calculate it using *i* and *j*. Look at that calculation for a bit to make sure it's correct. This is technically what the compiler does behind your back when you accessed the array using multidimensional notation.

12.7. Casting and promotion

Sometimes you have a type and you want it to be a different type. Here's a great example:

```
int main(void)
{
    int a = 5;
    int b = 10;
    float f;

    f = a / b;  /* calculate 5 divided by 10 */
    printf("%.2f\n", f);

    return 0;
}
```

And this prints:

```
0
```

...What? Five divided by 10 is zero? Since when? I'll tell you: since we entered the world of integer-only division. When you divide one `int` by another `int`, the result is an `int`, and any fractional part is thrown away. What do we do if we want the result to become a `float` somewhere along the way so that the result is correct?

Turns out, either integer (or both) in the divide can be made into a `float`, and then the result of the divide will be also be a `float`. So just change one and everything should work out.

“Get on with it! How do you cast?” Oh yeah--I guess I should actually do it. You might recall the cast from other parts of this guide, but just in case, we'll show it again:

```
f = (float)a / b;  /* calculate 5 divided by 10 */
```

Bam! There is is! Putting the new type in parens in front of the expression to be converted, and it magically becomes that type!

You can cast almost anything to almost anything else, and if you mess it up somehow, it's entirely your fault since the compiler will blindly do whatever you ask. `:-)`

12.8. Incomplete types

This topic is a little bit more advanced, but bear with it for a bit. An incomplete type is simply the declaration of the name of a particular `struct`, put there so that you can use pointers to the `struct` without actually knowing the fields stored therein. It most often comes up when people don't want to #include another header file, which can happen for a variety of different reasons.

For example, here we use a pointer to a type without actually having it defined anywhere in **`main()`**. (It is defined elsewhere, though.)

```
struct foo;  /* incomplete type!  Notice it's, well, incomplete. */

int main(void)
{
    struct foo *w;

    w = get_next_wombat(); /* grab a wombat */
    process_wombat(w);     /* use it somewhere */

    return 0;
}
```

I'm telling you this in case you find yourself trying to include a header that includes another header that includes the same header, or if your builds are taking forever because you're including too many headers, or...more likely you'll see an error along the lines of “cannot reference incomplete type”. This error means you've tried to do too much with the incomplete type (like you tried to dereference it or use a field in it), and you need to #include the right header file with the full complete declaration of the `struct`.

12.9. `void` pointers

Welcome to *THE VOID*! As Neo Anderson would say, “...Whoa.” What is this `void` thing?

Stop! Before you get confused, a `void` pointer isn't the same thing as a `void` return value from a function or a `void` argument list. I know that can be confusing, but there it is. Just wait until we talk about all the ways you can use the `static` keyword.

A `void` pointer is a *pointer to any type*. It is automatically cast to whatever type you assign into it, or copy from it. Why would you want to ever use such a thing? I mean, if you're going to dereference a pointer so that you can get to the original value, doesn't the compiler need to know what type the pointer is so that it can use it properly?

Yes. Yes, it does. Because of that, you cannot dereference a `void` pointer. It's against the law, and the C Police will be at your door faster than you can say Jack Robinson. Before you can use it, you have to cast it to another pointer type.

How on Valhalla is this going to be of any use then? Why would you even want a pointer you didn't know the type of?

The Specification: Write a function that can append pointers *of any type* to an array. Also write a function that can return a particular pointer for a certain index.

So in this case, we're going to write a couple useful little functions for storing off pointers, and returning them later. The function has be to *type-agnostic*, that is, it must be able to store pointers of any type. This is something of a fairly common feature to libraries of code that manipulate data--lots of them take `void` pointers so they can be used with any type of pointer the programmer might fancy.

Normally, we'd write a linked list or something to hold these, but that's outside the scope of this book. So we'll just use an array, instead, in this superficial example. Hmmm. Maybe I should write a beginning data structures book...

Anyway, the specification calls for two functions, so let's pound those puppies out right here:

```
#include <stdio.h>

void *pointer_array[10]; /* we can hold up to 10 void-pointers */
int index=0;

void append_pointer(void *p)
{
    pointer_array[index++] = p;
}

void *get_pointer(int i)
{
    return pointer_array[i];
}
```

Since we need to store those pointers somewhere, I went ahead and made a global array of them that can be accessed from within both functions. Also, I made a global index variable to remember where to store the next appended pointer in the array.

So check the code out for **`append_pointer()`** there. How is all that crammed together into one line? Well, we need to do two things when we append: store the data at the current index, and move the index to the next available spot. We copy the data using the assignment operator, and then notice that we use the *post-increment* operator (**++**) to increment the index. Remember what *post*-increment means? It means the increment is done *after* the rest of the expression is evaluated, including that assignment.

The other function, **`get_pointer`**, simply returns the `void*` at the specified index, *`i`*. What you want to watch for here is the subtle difference between the return types of the two functions. One of them is declared `void`, which means it doesn't return anything, and the other one is declared with a return type of `void*`, which means it returns a `void` pointer. I know, I know, the duplicate usage is a little troublesome, but you get used to it in a big hurry. Or else!

Finally, we have that code all written--now how can we actually use it? Let's write a **`main()`** function that will use these functions:

```
int main(void)
{
    char *s = "some data!";  /* s points to a constant string (char*) */
    int a = 10;
    int *b;

    char *s2;  /* when we call get_pointer(), we'll store them back here */
    int *b2;

    b = &a; /* b is a pointer to a */

    /* now let's store them both, even though they're different types */
    append_pointer(s);
    append_pointer(b);

    /* they're stored!  let's get them back! */
    s2 = get_pointer(0); /* this was at index 0 */
```

```
    b2 = get_pointer(1); /* this was at index 1 */

    return 0;
}
```

See how the pointer types are interchangable through the `void*`? C will let you convert the `void*` into any other pointer with impunity, and it really is up to you to make sure you're getting them back to the type they were originally. Yes, you can make mistakes here that crash the program, you'd better believe it. Like they say, “C gives you enough rope to hang yourself.”

12.10. `NULL` pointers

I think I have just enough time before the plane lands to talk about `NULL` when it comes to pointers.

`NULL` simply means a pointer to nothing. Sometimes it's useful to know if the pointer is valid, or if it needs to be initialized, or whatever. `NULL` can be used as a sentinel value for a variety of different things. Rememeber: it means “this pointer points to nothing”! Example:

```
int main(void)
{
    int *p = NULL;

    if (p == NULL) {
        printf("p is uninitialized!\n");
    } else {
        printf("p points to %d\n", *p);
    }

    return 0;
}
```

Note that pointers aren't preinitialized to `NULL` when you declare them--you have to explicitly do it. (No non-static local variables are preinitialized, pointers included.)

12.11. More Static

Modern technology has landed me safely here at LAX, and I'm free to continue writing while I wait for my plane to Ireland. Tell me you're not jealous at least on some level, and I won't believe you.

But enough about me; let's talk about programming. (How's that for a geek pick-up line? If you use it, do me a favor and don't credit me.)

You've already seen how you can use the `static` keyword to make a local variable persist between calls to its function. But there are other exciting completely unrelated uses for `static` that probably deserve to have their own keyword, but don't get one. You just have to get used to the fact that `static` (and a number of other things in C) have different meanings depending on context.

So what about if you declare something as `static` in the global scope, instead of local to a function? After all, global variables already persist for the life of the program, so `static` can't mean the same thing here. Instead, at the global scope, `static` means that the variable or function declared `static` *is only visible in this particular source file*, and cannot be referenced from other source files. Again, this definition of `static` only pertains to the global scope. `static` still means the same old thing in the local scope of the function.

You'll find that your bigger projects little bite-sized pieces themselves fit into larger bite-sized pieces (just like that picture of the little fish getting eaten by the larger fish being eaten by the fish that's larger, still.) When you have enough related smaller bite-sized pieces, it often makes sense to put them in their own source file.

I'm going to rip the example from the section on `void` pointers wherein we have a couple functions that can be used to store any types of pointers.

One of the many issues with that example program (there are all kinds of shortcomings and bugs in it) is that we've declared a global variable called *`index`*. Now, “index” is a pretty common word, and it's entirely likely that somewhere else in the project, someone will make up their own variable and name it the same thing as yours. This could cause all kinds of problems, not the least of which is they can modify your value of *`index`*, something that is very important to you.

One solution is to put all your stuff in one source file, and then declare *`index`* to be `static` global. That way, no one from outside your source file is allowed to use it. You are King! `static` is a way of keeping the implementation details of your portion of the code out of the hands of others. Believe me, if you let other people meddle in your code, they will do so with playful abandon! Big Hammer Smash!

So here is a quick rewrite of the code to be stuck it its own file:

```
/** file parray.c **/

static void *pointer_array[10]; /* now no one can see it except this file! */
static int index=0; /* same for this one! */

/* but these functions are NOT static, */
/* so they can be used from other files: */

void append_pointer(void *p)
{
    pointer_array[index++] = p;
}

void *get_pointer(int i)
{
    return pointer_array[i];
}

/** end of file parray.c **/
```

What would be proper at this point would be to make a file called *`parray.h`* that has the function prototypes for the two functions in it. Then the file that has **`main()`** in it can #include *`parray.h`* and use the functions when it is all linked together.

12.12. Typical Multifile Projects

Like I'm so fond of saying, projects generally grow too big for a single file, very similarly to how The Blob grew to be enormous and had to be defeated by Steve McQueen. Unfortunately, McQueen has already made his great escape to Heaven, and isn't here to help you with your code. Sorry.

So when you split projects up, you should try to do it in bite-sized modules that make sense to have in individual files. For instance, all the code responsible for calculating Fast Fourier Transforms (a mathematical construct, for those not in the know), would be a good candidate for its own source file. Or maybe all the code that controls a particular AI bot for a game could be in its own file. It's more of a guideline than a rule, but if something's not a least in some way related to what you have in a particular source file already, maybe it should go elsewhere. A perfect illustrative question for this scenario might be, “What is the 3D rendering code doing in the middle of the sound driver code?”

When you do move code to its own source file, there is almost always a header file that you should write to go along with it. The code in the new source file (which will be a bunch of functions) will need to have prototypes and other things made visible to the other files so they can be used. The way the other source files use other files is to $#include their header files.

So for example, let's make a small set of functions and stick them in a file called *simplemath.c*:

```
/** file simplemath.c **/

int plusone(int a)
{
    return a+1;
}

int minusone(int a)
{
    return a-1;
}

/** end of file simplemath.c **/
```

A couple simple functions, there. Nothing too complex, yes? But by themselves, they're not much use, and for other people to use them we need to distribute their prototypes in a header file. Get ready to absorbe this...you should recognize the prototypes in there, but I've added some new stuff:

```
/** file simplemath.h **/

#ifndef _SIMPLEMATH_H_
#define _SIMPLEMATH_H_

/* here are the prototypes: */

int plusone(int a);
int minusone(int a);

#endif

/** end of file simplemath.h **/
```

Icky. What is all that #ifndef stuff? And the #define and the #endif? They are *boilerplate code* (that is, code that is more often than not stuck in a file) that prevents the header file from being included multiple times.

The short version of the logic is this: if the symbol *_SIMPLEMATH_H_* isn't defined then define it, and then do all the normal header stuff. Else if the symbol *_SIMPLEMATH_H_ is* already defined, do nothing. In this way, the bulk of the header file is included only once for the build, no matter how many other files try to include it. This is a good idea, since at best it's a redundant waste of time to re-include it, and at worst, it can cause compile-time errors.

Well, we have the header and the source files, and now it's time to write a file with **main()** in it so that we can actually use these things:

```
/** file main.c **/

#include "simplemath.h"

int main(void)
{
    int a = 10, b;

    b = plusone(a);  /* make that processor work! */
```

```
    return 0;
}

/** end of file main.c **/
```

Check it out! We used double-quotes in the #include instead of angle brackets! What this tells the preprocessor to do is, “include this file from the current directory instead of the standard system directory.” I'm assuming that you're putting all these files in the same place for the purposes of this exercise.

Recall that including a file is exactly like bringing it into your source at that point, so, as such, we bring the prototypes into the source right there, and then the functions are free to be used in **main()**. Huzzah!

One last question! How do we actually build this whole thing? From the command line:

```
$ cc -o main main.c simplemath.c
```

You can lump all the sources on the command line, and it'll build them together, nice and easy.

12.13. The Almighty C Preprocessor

Remember back about a million years ago when you first started reading this guide and I mentioned something about spinach? That's right--you remember how spinach relates to the whole computing process?

Of course you don't remember. I just made it up just now; I've never mentioned spinach in this guide. I mean, c'mon. What does spinach have to do with anything? Sheesh!

Let me steer you to a less leafy topic: the C preprocessor. As the name suggests, this little program will process source code before the C compiler sees it. This gives you a little bit more control over what gets compiled and how it gets compiled.

You've already seen one of the most common preprocessor directives: #include. Other sections of the guide have touched upon various other ones, but we'll lay them all out here for fun.

12.13.1. #include

The well-known #include directive pulls in source from another file. This other file should be a header file in virtually every single case.

On each system, there are a number of standard include files that you can use for various tasks. Most popularly, you've seen *stdio.h* used. How did the system know where to find it? Well, each compiler has a set of directories it looks in for header files when you specify the file name in angle brackets. (On Unix systems, it commonly searches the */usr/include* directory.)

If you want to include a file from the same directory as the source, use double quotes around the name of the file. Some examples:

```
/* include from the system include directory: */

#include <stdio.h>
#include <sys/types.h>

/* include from the local directory: */

#include "mutants.h"
#include "fishies/halibut.h"
```

As you can see from the example, you can also specify a *relative path* into subdirectories out of the main directory. (Under Unix, again, there is a file called *types.h* in the directory */usr/include/sys*.)

12.13.2. #define

The #define is one of the more powerful C preprocessor directives. With it you can declare constants to be substituted into the source code before the compiler even sees them. Let's say you have a lot of constants scattered all over your program and each number is *hard-coded* (that is, the number is written explicitly in the code, like “2”).

Now, you thought it was a constant when you wrote it because the people you got the specification swore to you up and down on pain of torture that the number would be “2”, and it would never change in 100 million years so strike them blind right now.

Hey--sounds good. You even have someone to blame if it did change, and it probably won't anyway since they seem so sure.

Don't be a fool.

The spec will change, and it will do so right after you have put the number “2” in approximately three hundred thousand places throughout your source, they're going to say, “You know what? Two just isn't going to cut it--we need three. Is that a hard change to make?”

Blame or not, you're going to be the one that has to change it. A good programmer will realize that hard-coding numbers like this isn't a good idea, and one way to get around it is to use a #define directive. Check out this example:

```
#define PI 3.14159265358979  /* more pi than you can handle */

int main(void)
{
    float r =10.0;

    printf("pi: %f\n", PI);
    printf("pi/2: %f\n", PI/2);
    printf("area: %f\n", PI*r*r);
    printf("circumference: %f\n", 2*PI*r);

    return 0;
}
```

(Typically #defines are all capitals, by convention.) So, hey, we just printed that thing out as if it was a float. Well, it *is* a float. Remember--the C preprocessor substitutes the value of *PI* *before the compiler even sees it*. It is just as if you had typed it there yourself.

Now let's say you've used *PI* all over the place and now you're just about ready to ship, and the designers come to you and say, “So, this whole pi thing, um, we're thinking it needs to be four instead of three-point-one-whatever. Is that a hard change?”

No problem in this case, no matter how deranged the change request. All you have to do is change the one #define at the top, and it's therefore automatically changed all over the code when the C preprocessor runs through it:

```
#define PI 4.0  /* whatever you say, boss */
```

Pretty cool, huh. Well, it's perhaps not as good as to be “cool”, but you have not yet witnessed the destructive power of this fully operational preprocessor directive! You can actually use #define to write little *macros* that are like miniature functions that the preprocessor evaluates, again, before the C compiler sees the code. To make a macro like this, you give an argument list (without types, because the preprocessor knows nothing about types, and then you list how that is to be used. For instance, if we want a macro that evaluates to a number you pass it times 3490, we could do the following:

```
#define TIMES3490(x) ((x)*3490)    /* no semicolon, notice! */

void evaluate_fruit(void)
{
    printf("40 * 3490 = %d\n", TIMES3490(40));
}
```

In that example, the preprocessor will take the macro and *expand* it so that it looks like this to the compiler:

```
void evaluate_fruit(void)
{
    printf("40 * 3490 = %d\n", ((40)*3490));
}
```

(Actually the preprocessor can do basic math, so it'll probably reduce it directly to "139600". But this is my example and I'll do as I please!)

Now here's a question for you: are you taking it on blind faith that you need all those parenthesis in the macro, like you're some kind of LISP superhero, or are you wondering, "Why am I wasting precious moments of my life to enter all these parens?"

Well, you *should* be wondering! It turns out there are cases where you can really generate some *evil* code with a macro without realizing it. Take a look at this example which builds without a problem:

```
#define TIMES3490(x) x*3490

void walrus_discovery_unit(void)
{
    int tuskcount = 7;

    printf("(tuskcount+2) * 3490 = %d\n", TIMES3490(tuskcount + 2));
}
```

What's wrong here? We're calculating `tuskcount+2`, and then passing that through the **`TIMES3490()`** macro. But look at what it expands to:

```
printf("(tuskcount+2) * 3490 = %d\n", tuskcount+2*3490);
```

Instead of calculating `(tuskcount+2)*3490`, we're calculating `tuskcount+(2*3490)`, because multiplication comes before addition in the order of operations! See, adding all those extra parens to the macro prevents this sort of thing from happening. So programmers with good practices will automatically put a set of parens around each usage of the parameter variable in the macro, as well as a set of parens around the outside of the macro itself.

12.13.3. #if and #ifdef

There are some *conditionals* that the C preprocessor can use to discard blocks of code so that the compiler never sees them. The #if directive operates like the C if, and you can pass it an expression to be evaluated. It is most common used to block off huge chunks of code like a comment, when you don't want it to get built:

```
void set_hamster_speed(int warpfactor)
{
#if 0
    uh this code isn't written yet.  someone should really write it.
#endif
```

```
}
```

You can't nest comments in C, but you can nest #if directives all you want, so it can be very helpful for that.

The other if-statement, #ifdef is true if the subsequent macro is already defined. There's a negative version of this directive called #ifndef ("if not defined"). #ifndef is very commonly used with header files to keep them from being included multiple times:

```
/** file aardvark.h **/

#ifndef _AARDVARK_H_
#define _AARDVARK_H_

int get_nose_length(void);
void set_nose_length(int len);

#endif

/** end of file aardvark.h **/
```

The first time this file is included, *_AARDVARK_H_* is not yet defined, so it goes to the next line, and defines it, and then does some function prototypes, and you'll see there at the end, the whole #if-type directive is culminated with an #endif statement. Now if the file is included again (which can happen when you have a lot of header files are including other header files *ad infinininininini*--cough!), the macro *_AARDVARK_H_* will already be defined, and so the #ifndef will fail, and the file up to the #endif will be discarded by the preprocessor.

Another extremely useful thing to do here is to have certain code compile for a certain platform, and have other code compile for a different platform. Lots of people like to build software with a macro defined for the type of platform they're on, such as *LINUX* or *WIN32*. And you can use this to great effect so that your code will compile and work on different types of systems:

```
void run_command_shell(void)
{
#ifdef WIN32
    system("COMMAND.EXE");
#elifdef LINUX
    system("/bin/bash");
#else
#error We don't have no steenkin shells!
#endif
}
```

A couple new things there, most notable #elifdef. This is the contraction of "else ifdef", which must be used in that case. If you're using #if, then you'd use the corresponding #elif.

Also I threw in an #error directive, there. This will cause the preprocessor to bomb out right at that point with the given message.

12.14. Pointers to pointers

You've already seen how you can have a pointer to a variable...and you've already seen how a pointer *is* a variable, so is it possible to have a *pointer to a pointer*?

No, it's not.

I'm kidding--of course it's possible. Would I have this section of the guide if it wasn't?

There are a few good reasons why we'd want to have a pointer to a pointer, and we'll give you the simple one first: you want to pass a pointer as a parameter to a function, have the function modify it, and have the results reflected back to the caller.

Note that this is exactly the reason why we use pointers in function calls in the first place: we want the function to be able to modify the thing the pointer points to. In this case, though, the thing we want it to modify is another pointer. For example:

```
void get_string(int a, char **s)
{
    switch(a) {
        case 0:
            *s = "everybody";
            break;

        case 1:
            *s = "was";
            break;

        case 2:
            *s = "kung-foo fighting";
            break;

        default:
            *s = "errrrrrnt!";
    }
}

int main(void)
{
    char *s;

    get_string(2, &s);

    printf("s is \"%s\"\n", s);  /* 's is "kung-foo fighting"' */

    return 0;
}
```

What we have, above, is some code that will deliver a string (pointer to a `char`) back to the caller via pointer to a pointer. Notice that we pass the *address of* the pointer *s* in **`main()`**. This gives the function **`get_string()`** a pointer to *s*, and so it can dereference that pointer to change what it is pointing at, namely *s* itself.

There's really nothing mysterious here. You have a pointer to a thing, so you can dereference the pointer to change the thing. It's just like before, except for that fact that we're operating on a pointer now instead of just a plain base type.

What else can we do with pointers to pointers? You can dynamically make a construction similar to a two-dimensional array with them. The following example relies on your knowledge that the function call **`malloc()`** returns a chunk of sequential bytes of memory that you can use as you will. In this case, we'll use them to create a number of `char*`s. And we'll have a pointer to that, as well, which is therefore of type `char**`.

```
int main(void)
{
    char **p;

    p = malloc(sizeof(char*) * 10);  // allocate 10 char*s

    return 0;
}
```

Swell. Now what can we do with those? Well, they don't point to anything yet, but we can call **`malloc()`** for each of them in turn and then we'll have a big block of memory we can store strings in.

```
int main(void)
{
    char **p;
    int i;

    p = malloc(sizeof(char*) * 10);  // allocate 10 char*s-worth of bytes

    for(i = 0; i < 10; i++) {
        *(p+i) = malloc(30); // 30 bytes for each pointer

        // alternatively we could have written, above:
        //   p[i] = malloc(30);
        // but we didn't.

        sprintf(*(p+i), "this is string #%d", i);
    }

    for(i = 0; i < 10; i++) {
        printf("%d: %s\n", i, p[i]); // p[i] same as *(p+i)
    }

    return 0;
}
```

Ok, as you're probably thinking, this is where things get completely wacko-jacko. Let's look at that second **`malloc()`** line and dissect it one piece at a time.

You know that *`p`* is a pointer to a pointer to a `char`, or, put another way, it's a pointer to a `char*`. Keep that in mind.

And we know this `char*` is the first of a solid block of 10, because we just **`malloc()`**'d that many before the for loop. With that knowledge, we know that we can use some pointer arithmetic to hop from one to the next. We do this by adding the value of *`i`* onto the `char**` so that when we dereference it, we are pointing at the next `char*` in the block. In the first iteration of the loop *`i`* is zero, so we're just referring to the first `char*`.

And what do we do with that `char*` once we have it? We point it at the return value of **`malloc()`** which will point it at a fresh ready-to-use 30 bytes of memory.

And what do we use that memory for (sheesh, this could go on forever!)--well, we use a variant of **`printf()`** called **`sprintf()`** that writes the result into a string instead of to the console.

And there you have it. Finally, for fun, we print out the results using array notation to access the strings instead of pointer arithmetic.

12.15. Pointers to Functions

You've completely mastered all that pointer stuff, right? I mean, you are the *Pointer Master*! No, really, I insist!

So, with that in mind, we're going to take the whole pointer and address idea to the next phase and learn a little bit about the machine code that the compiler produces. I know this seems like it has nothing to do with this section, Pointers to Functions, but it's background that will only make you stronger. (Provided, that is, it doesn't kill you first. Admittedly, the chances of death from trying to understand this section are slim, but you might want to read it in a padded room just as a precautionary measure.)

Long ago I mentioned that the compiler takes your C source code and produces *machine code* that the processor can execute. These machine code instructions are small (taking between one and four bytes or memory, typically, with optionally up to, say, 32 bytes of arguments per instruction--these numbers vary depending on the processor in question). This isn't so important as the fact that these instructions have to be stored somewhere. Guess where.

You thought that was a rhetorical command, but no, I really do want you to guess where, generally, the instructions are stored.

You have your guess? Good. Is it animal, vegetable, or mineral? Can you fly in it? Is it a rocketship? Yay!

But, cerebral digression aside, yes, you are correct, the instructions are stored in memory, just like variables are stored in memory. Instructions themselves have addresses, and a special variable in the CPU (generally known as a “register” in CPU-lingo) points to the address of the currently executing instruction.

Whatwhat? I said “points to” and “address-of”! Suddenly we have a connection back to pointers and all that...familiar ground again. But what did I just say? I said: instructions are held in addresses, and, therefore, you have have a pointer to a block of instructions. A block of instructions in C is held in a function, and, therefore, you can have a pointer to a function. *Voila*!

Ok, so if you have a function, how do you get the address of the function? Yes, you can use the `&`, but most people don't. It's similar to the situation with arrays, where the name of the array without square brackets is a pointer to the first element in the array; the name of the function without parens is a pointer to the first instruction in the function. That's the easy part.

The hard part is declaring a variable to be of type “pointer to function”. It's hard because the syntax is funky:

```
// declare p as a pointer to a function that takes two int
// parameters, and returns a float:

float (*p)(int, int);
```

Again, note that this is a *declaration of a pointer* to a function. It doesn't yet point to anything in particular. Also notice that you don't have to put dummy parameter names in the declaration of the pointer variable. All right, let's make a function, point to it, and call it:

```
int deliver_fruit(char *address, float speed)
{
    printf("Delivering fruit to %s at speed %.2f\n", address, speed);

    return 3490;
}

int main(void)
{
    int (*p)(char*,float);  // declare a function pointer variable

    p = deliver_fruit; // p now points to the deliver_fruit() function

    deliver_fruit("My house", 5280.0); // a normal call

    p("My house", 5280.0); // the same call, but using the pointer

    return 0;
}
```

What the heck good is this? The usual reasons are these:

- You want to change what function is called at runtime.

- You have a big array of data including pointers to functions.
- You don't know the function at compile-time; maybe it's in a shared library that you load a runtime and query to find a function, and that query returns a pointer to the function. I know this is a bit beyond the scope of the section, but bear with me.

For example, long ago a friend of mine and I wrote a program that would simulate a bunch of creatures running around a grid. The creatures each had a `struct` associated with them that held their position, health, and other information. The `struct` also held a pointer to a function that was their behavior, like this:

```
struct creature {
    int xpos;
    int ypos;
    float health;
    int (*behavior)(struct useful_data*);
};
```

So for each round of the simulation, we'd walk through the list of creatures and call their behavior function (passing a pointer to a bunch of useful data so the function could see other creatures, know about itself, etc.) In this way, it was easy to code bugs up as having different behaviors.

Indeed, I wrote a creature called a "brainwasher" that would, when it got close to another creature, change that creature's behavior pointer to point to the brainwasher's behavior code! Of course, it didn't take long before they were all brainwashers, and then starved and cannibalized themselves to death. Let that be a lesson to you.

12.16. Variable Argument Lists

Ever wonder, in your spare time, while you lay awake at night thinking about the C Programming Language, how functions like **`printf()`** and **`scanf()`** seem to take an arbitrary number of arguments and other functions take a specific number? How do you even write a function prototype for a function that takes a variable number of arguments?

(Don't get confused over terminology here--we're not talking about variables. In this case, "variable" retains its usual boring old meaning of "an arbitrary number of".)

Well, there are some little tricks that have been set up for you in this case. Remember how all the arguments are pushed onto the stack when passed to a function? Well, some macros have been set up to help you walk along the stack and pull arguments off one at a time. In this way, you don't need to know at compile-time what the argument list will look like--you just need to know how to parse it.

For instance, let's write a function that averages an arbitrary number of positive numbers. We can pull numbers off the stack one at a time and average them all, but we need to know when to stop pulling stuff off the stack. One way to do this is to have a sentinel value that you watch for--when you hit it, you stop. Another way is to put some kind of information in the mandatory first argument. Let's do option B and put the count of the number of arguments to be averaged in as the first argument to the function.

Here's the prototype for the function--this is how we declare a variable argument list. The first argument (at least) must be specified, but that's all:

```
float average(int count, ...);
```

It's the magical "`...`" that does it, see? This lets the compiler know that there can be more arguments after the first one, but doesn'r require you to say what they are. So *this* is how we are able to pass many or few (but at least one, the first argument) arguments to the function.

But if we don't have names for the variables specified in the function header, how do we use them in the function? Well, aren't we the demanding ones, actually wanting to *use* our function! Ok, I'll tell you!

There is a special type declared in the header *stdarg.h* called `va_list`. It holds data about the stack and about what arguments have been parsed off so far. But first you have to tell it where the stack for this function starts, and fortunately we have a variable right there at the beginning of our **`average()`** function: *a*.

We operate on our `va_list` using a number of preprocessor macros (which are like mini-functions if you haven't yet read the section on macros.) First of all, we use **`va_start()`** to tell our `va_list` where the stack starts. Then we use **`va_arg()`** repeatedly to pull arguments off the stack. And finally we use **`va_end()`** to tell our `va_list` that we're done with it. (The language specification says we *must* call **`va_end()`**, and we must call it in the same function from which we called **`va_start()`**. This allows the compiler to do any cleanup that is necessary, and keeps the Vararg Police from knocking on your door.

So an example! Let's write that **`average()`** function. Remember: **`va_start()`**, **`va_arg()`**, **`va_arg()`**, **`va_arg()`**, etc., and then **`va_end()`**!

```
float average(int count, ...)
{
    float ave = 0;
    int i;
    va_list args; // here's our va_list!

    va_start(args, count); // tell it the stack starts with "count"

    // inside the while(), pull int args off the stack:

    for(i = 0; i < count; i++) {
        int val = va_arg(args, int); // get next int argument
        ave += (float)val;  // cast the value to a float and add to total
    }

    va_end(args);  // clean this up

    return ave / count; // calc and return the average
}
```

So there you have it. As you see, the **`va_arg()`** macro pulls the next argument off the stack of the given type. So you have to know in advance what type the thing is. We know for our **`average()`** function, all the types are `int`s, so that's ok. But what if they're different types mixed all together? How do you tell what type is coming next?

Well, if you'll notice, this is exactly what our old friend **`printf()`** does! It knows what type to call **`va_arg()`** with, since it says so right in the format string.

12.16.1. `vprintf()` and its ilk

There are a number of functions that helpfully accept a `va_list` as an argument that you can pass. This enables you to wrap these functions up easily in your own functions that take a variable number of arguments themselves. For instance:

Assignment: Implement a version of **`printf()`** called **`timestamp_printf()`** that works *exactly* like **`printf()`** except it prints the time followed by a newline followed by the data output specified by the **`printf()`**-style format string.

Holy cow! At first glance, it looks like you're going to have to implement a clone of **`printf()`** just to get a timestamp out in front of it! And **`printf()`** is, as we say in the industry, “nontrivial”! See you next year!

Wait, though--wait, wait...there *must* be a way to do it easily, or this author is complete insane to give you this assignment, and that couldn't be. Fruit! Where is my cheese!? Blalalauugh!!

Ahem. I'm all right, really, Your Honor. I'm looking into my crystal ball and I'm seeing...a type `va_list` in your future. In fact, if we took our variable argument list, processed it with **`va_start()`** and got our `va_list` back, we could, if such a thing existed, just pass it to an already-written version of **`printf()`** that accepted just that thing.

Welcome to the world of **`vprintf()`**! It does exactly that, by Jove! Here's a lovely prototype:

```
int vprintf(const char *format, va_list args);
```

All righty, so what building blocks do we need for this assignment? The spec says we need to do something just like **`printf()`**, so our function, like **`printf()`** is going to accept a format string first, followed by a variable number of arguments, something like this:

```
int timestamp_printf(char *format, ...);
```

But before it prints its stuff, it needs to output a timestamp followed by a newline. The exact format of the timestamp wasn't specified in the assignment, so I'm going to assume something in the form of "weekday month day hh:mm:ss year". By amazing coincidence, a function exists called **`ctime()`** that returns a string in exactly that format, given the current system time.

So the plan is to print a timestamp, then take our variable argument list, run it through **`va_start`** to get a `va_list` out of it, and pass that `va_list` into **`vprintf()`** and let it work its already-written **`printf()`** magic. And...GO!

```
#include <stdio.h>
#include <stdarg.h>
#include <time.h>  // for time() and ctime();

int timestamp_printf(char *format, ...)
{
    va_list args;
    time_t system_time;
    char *timestr;
    int return_value;

    system_time = time(NULL); // system time in seconds since epoch
    timestr = ctime(&system_time); // ready-to-print timestamp

    // print the timestamp:
    printf("%s", timestr); // timestr has a newline on the end already

    // get our va_list:
    va_start(args, format);

    // call vprintf() with our arg list:
    return_value = vprintf(format, args);

    // done with list, so we have to call va_end():
    va_end(args);

    // since we want to be *exactly* like printf(), we have saved its
    // return value, and we'll pass it on right here:
    return return_value;
}
```

```
int main(void)
{
    // example call:
    timestamp_printf("Brought to you by the number %d\n", 3490);

    return 0;
}
```

And there you have it! Your own little **`printf()`**-like functionality!

Now, not every function has a "v" in front of the name for processing variable argument lists, but most notably all the variants of **`printf()`** and **`scanf()`** do, so feel free to use them as you see fit!

TODO order of operations, arrays of pointers to functions

13. 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 reading and 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:

`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.

13.1. `fopen()`

Opens a file for reading or writing

Prototypes

```
#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.

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

`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.

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;

    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

    return 0;
}
```

See Also

`fclose()`
`freopen()`

13.2. `freopen()`

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

Prototypes

```
#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), **`fopen()`** 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"

    return 0;
}
```

See Also

`fclose()`
`fopen()`

13.3. `fclose()`

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

Prototypes

```
#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()`

13.4. `printf()`, `fprintf()`

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

Prototypes

```
#include <stdio.h>

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

Description

These functions print formatted strings to a file (that is, a `FILE*` you likely got from **`fopen()`**), or to the console (which is usually itself just a special file, right?)

The **`printf()`** function is legendary as being one of the most flexible outputting systems ever devisied. 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.

Here are the most common percent codes:

`%d`
: Print the next argument as a signed decimal number, like `3490`. The argument printed this way should be an `int`.

`%f`
: Print the next argument as a signed floating point number, like `3.14159`. The argument printed this way should be a `float`.

`%c`
: Print the next argument as a character, like `'B'`. The argument printed this way should be a `char`.

`%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 percent codes 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 there are still more modifiers you can put in after the percent and before the field width:

`0`
: This was already mentioned above. It pads the spaces before a number with zeros, e.g. `"%05d"`.

`-`
: This was also already mentioned above. It causes the value to be left-justified in the field, e.g. `"%-5d"`.

`' '` (space)
: This prints a blank space before a positive number, so that it will line up in a column along with negative numbers (which have a negative sign in front of them). `"% d"`.

`+`
: Always puts a + sign in front of a number that you print so that it will line up in a column along with negative numbers (which have a negative sign in front of them). `"%+d"`.

`#`
: This causes the output to be printed in a different form than normal. The results vary based on the specifier used, but generally, hexidecimal output (`"%x"`) gets a `"0x"` prepended to the output, and octal output (`"%o"`) gets a `"0"` prepended to it. These are, if you'll notice, how such numbers are represented in C source. Additionally, floating point numbers, when printed with this `#` modified, will print a trailing decimal point even if the number has no fractional part. Example: `"%#x"`.

Now, I know earlier I promised the rest of the format specifiers...so ok, here they are:

`%i`
: Just like `%d`, above.

`%o`
: Prints the integer number out in octal format. Octal is a base-eight number representation scheme invented on the planet Krylon where all the inhabitants have only eight fingers.

`%u`
: Just like `%d`, but works on `unsigned int`s, instead of `int`s.

`%x` or `%X`
: Prints the `unsigned int` argument in hexidecimal (base-16) format. This is for people with 16 fingers, or people who are simply addicted hex, like you should be. Just try it! `"%x"` prints the hex digits in lowercase, while `"%X"` prints them in uppercase.

`%F`
: Just like "%f", except any string-based results (which can happen for numbers like infinity) are printed in uppercase.

`%e` or `%E`
: Prints the `float` argument in exponential (scientific) notation. This is your classic form similar to "three times 10 to the 8th power", except printed in text form: "`3e8`". (You see, the "`e`" is read "times 10 to the".) If you use the `"%E"` specifier, the the exponent "e" is written in uppercase, a la "`3E8`".

`%g` or `%G`
: Another way of printing `double`s. In this case the precision you specific tells it how many significant figures to print.

`%p`
: Prints a pointer type out in hex representation. In other words, the address that the pointer is pointing to is printed. (Not the value in the address, but the address number itself.)

`%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.

So let's recap what we have here. We have a format string in the form:

```
"%[modifier][fieldwidth][.precision][lengthmodifier][formatspecifier]"
```

Modifier is like the `"-"` for left justification, the field width is how wide a space to print the result in, the precision is, for `float`s, how many decimal places to print, and the format specifier is like `%d`.

That wraps it up, except what's this "lengthmodifier" I put up there?! Yes, just when you thought things were under control, I had to add something else on there. Basically, it's to tell **`printf()`** in more detail what size the arguments are. For instance, `char`, `short`, `int`, and `long` are all integer types, but they all use a different number of bytes of memory, so you can't use plain old "`%d`" for *all* of them, right? How can **`printf()`** tell the difference?

The answer is that you tell it explicitly using another optional letter (the length modifier, this) before the type specifier. If you omit it, then the basic types are assumed (like `%d` is for `int`, and `%f` is for `float`).

Here are the format specifiers:

`h`
: Integer referred to is a `short` integer, e.g. "`%hd`" is a `short` and "`%hu`" is an `unsigned short`.

`l` ("ell")
: Integer referred to is a `long` integer, e.g. "`%ld`" is a `long` and "`%lu`" is an `unsigned long`.

`hh`
: Integer referred to is a `char` integer, e.g. "`%hhd`" is a `char` and "`%hhu`" is an `unsigned char`.

`ll` ("ell ell")
: Integer referred to is a `long long` integer, e.g. "`%lld`" is a `long long` and "`%llu`" is an `unsigned long long`.

I know it's hard to believe, but there might be *still more* format and length specifiers on your system. Check your manual for more information.

Return Value

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()`**, **`vfprintf()`**, **`vsprintf()`**

13.5. `scanf()`, `fscanf()`

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

Prototypes

```
#include <stdio.h>

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

Description

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 percent-codes you can put in the format string:

`%d`
: Reads an integer to be stored in an `int`. This integer can be signed.

`%f` (`%e`, `%E`, and `%g` are equivalent)
: Reads a floating point number, to be stored in a `float`.

`%s`
: Reads a string. This will stop on the first whitespace character reached, or at the specified field width (e.g. "%10s"), whichever comes first.

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!

`%u`
: Reads an unsigned integer to be stored in an `unsigned int`.

`%x` (`%X` is equivalent)
: Reads an unsigned hexidecimal integer to be stored in an `unsigned int`.

`%o`
: Reads an unsigned octal integer to be stored in an `unsigned int`.

`%i`
: Like `%d`, except you can preface the input with "0x" if it's a hex number, or "0" if it's an octal number.

`%c`
: Reads in a character to be stored in a `char`. If you specify a field width (e.g. "`%12c`", it will read that many characters, so make sure you have an array that large to hold them.

`%p`
: Reads in a pointer to be stored in a `void*`. The format of this pointer should be the same as that which is outputted with **`printf()`** and the "`%p`" format specifier.

`%n`
: Reads nothing, but will store the number of characters processed so far into the next `int` parameter in the argument list.

`%%`
: Matches a literal percent sign. No conversion of parameters is done. This is simply how you get a standalone percent sign in your string without **`scanf()`** trying to do something with it.

`%[`
: This is about the weirdest format specifier there is. It allows you to specify a set of characters to be stored away (likely in an array of `char`s). 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: `%[A-C-]`.

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

So those are the basics! Phew! There's a lot of stuff to know, but, like I said, a few of these format specifiers are common, and the others are pretty rare.

Got it? Now we can go onto the next--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!

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:

`h`
: The value to be parsed is a `short int` or `short unsigned`. Example: `%hd` or `%hu`.

`l`
: The value to be parsed is a `long int` or `long unsigned`, or `double` (for `%f` conversions.) Example: `%ld`, `%lu`, or `%lf`.

`L`

The value to be parsed is a `long long` for integer types or `long double` for `float` types. Example: `%Ld`, `%Lu`, or `%Lf`.

`*`

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. Example: `%*d`.

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

```
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()`**

13.6. `gets()`, `fgets()`

Read a string from console or file

Prototypes

```
#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()`**.

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()`**:

```

char s[100];
gets(s);  // 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, like so:

```

char *remove_newline(char *s)
{
    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 **`fgets()`** 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()**

13.7. `getc()`, `fgetc()`, `getchar()`

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

Prototypes

```
#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);

    return 0;
}
```

See Also

13.8. puts(), fputs()

Write a string to the console or to a file.

Prototypes

```
#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);

    return 0;
}
```

See Also

13.9. `putc()`, `fputc()`, `putchar()`

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

Prototypes

```
#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

    return 0;
}
```

See Also

13.10. `fseek()`, `rewind()`

Position the file pointer in anticipition of the next read or write.

Prototypes

```
#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:

`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()`**

13.11. `ftell()`

Tells you where a particular file is about to read from or write to.

Prototypes

```
#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()`**

13.12. `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.

Prototypes

```
#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()`**

13.13. `ungetc()`

Pushes a character back into the input stream.

Prototypes

```
#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()

13.14. `fread()`

Read binary data from a file.

Prototypes

```
#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.

(As for the prototype, what is this `size_t` you see floating around? It's short for “size type” which is a data type defined to hold the size of something. Great--would I stop beating around the bush already and give you the straight story?! Ok, `size_t` is probably an `int`.)

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]);

    return 0;
}
```

See Also

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

13.15. `fwrite()`

Write binary data to a file.

Prototypes

```
#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);

    return 0;
}
```

See Also

`fopen()`, **`fread()`**

13.16. `feof()`, `ferror()`, `clearerr()`

Determine if a file has reached end-of-file or if an error has occurred.

Prototypes

```
#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);

    return 0;
}
```

See Also

`fopen()`, **`fread()`**

13.17. `perror()`

Print the last error message to *`stderr`*

Prototypes

```
#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`* 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 catch is that different systems define different values for *`errno`*, so it's not very portable. The standard only defines a few math-related values, and not others. You'll have to check your local man-pages for what works on your system.

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");
        }
    }

    return 0;
```

```
}
```

And the output is:

```
fseek: Illegal seek
fseek again, EBADF: Bad file descriptor
```

See Also

feof(), **ferror()**, **clearerr()**

13.18. **`remove()`**

Delete a file

Prototypes

```
#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()`

13.19. `rename()`

Renames a file and optionally moves it to a new location

Prototypes

```
#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()`

13.20. `tmpfile()`

Create a temporary file

Prototypes

```
#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)

    return 0;
}
```

See Also

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

13.21. `tmpnam()`

Generate a unique name for a temporary file

Prototypes

```
#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.

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()`

13.22. setbuf(), setvbuf()

Configure buffering for standard I/O operations

Prototypes

```
#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:

`_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()

13.23. `fflush()`

Process all buffered I/O for a stream right now

Prototypes

```
#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);
    }

    return 0;
}
```

See Also

setbuf(), **setvbuf()**

14. 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.

14.1. strlen()

Returns the length of a string.

Prototypes

```
#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 characters in the 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

14.2. `strcmp()`, `strncmp()`

Compare two strings and return a difference.

Prototypes

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Description

Both these functions compare two strings. **`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*.

For the most part, people just check to see if the return value is zero or not, because, more often than not, people are only curious if strings are the same.

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 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()`**

14.3. `strcat()`, `strncat()`

Concatenate two strings into a single string.

Prototypes

```
#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()

14.4. `strchr()`, `strrchr()`

Find a character in a string.

Prototypes

```
#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 occurance 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 occurance 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 occurance 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 occurances 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

14.5. `strcpy()`, `strncpy()`

Copy a string

Prototypes

```
#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
// 01234567890123456789012345
```

See Also

memcpy(), **strcat()**, **strncat()**

14.6. `strspn()`, `strcspn()`

Return the length of a string consisting entirely of a set of characters, or of not a set of characters.

Prototypes

```
#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 lenght 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()`**

14.7. `strstr()`

Find a string in another string.

Prototypes

```
#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 occurance 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()`**

14.8. `strtok()`

Tokenize a string.

Prototypes

```
#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()`**

15. 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 GOD!

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.

`M_E`
: e

`M_LOG2E`
: log_2 e

`M_LOG10E`
: log_10 e

`M_LN2`
: log_e 2

`M_LN10`
: log_e 10

`M_PI`
: pi

`M_PI_2`
: pi/2

`M_PI_4`
: pi/4

`M_1_PI`
: 1/pi

`M_2_PI`
: 2/pi

`M_2_SQRTPI`
: 2/sqrt(pi)

`M_SQRT2`
: sqrt(2)

`M_SQRT1_2`
: 1/sqrt(2)

15.1. `sin()`, `sinf()`, `sinl()`

Calculate the sine of a number.

Prototypes

```
#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()`**

15.2. `cos()`, `cosf()`, `cosl()`

Calculate the cosine of a number.

Prototypes

```
#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()`**

15.3. `tan()`, `tanf()`, `tanl()`

Calculate the tangent of a number.

Prototypes

```
#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()`**

15.4. asin(), asinf(), asinl()

Calculate the arc sine of a number.

Prototypes

```
#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()**

15.5. `acos()`, `acosf()`, `acosl()`

Calculate the arc cosine of a number.

Prototypes

```
#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()`**

15.6. atan(), atanf(), atanl(), atan2(), atan2f(), atan2l()

Calculate the arc tangent of a number.

Prototypes

```
#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()**

15.7. `sqrt()`

Calculate the square root of a number

Prototypes

```
#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()`

16. Complex Numbers

17. Time Library