The C Programming Language

Preface:

This document covers as much as I could humanly write about the C programming language. I’ve been programming for approximately 5 years or so, mostly in C, although I started writing this document way back when I first learned C, so please forgive for any misinformation that got overlooked. I try to update the document any time I discover something new about the language or discover that my presuppositions were wrong. I hope you get something out of reading this, as that is all I could hope for after putting many hours into it. Happy programming!

Intro:

The C programming language is primarily attributed to a programmer named Dennis Ritchie. Most programmers tend to associate C with operating systems – a) because it is a good language for interfacing with low level hardware, and b) because of the success of Unix/Linux, which were both written in C. The Unix operating system was created in the 1960s in Bell Laboratories (Bell Labs for short). Unix was originally written in assembly language, but in 1973 Unix was rewritten by Dennis Ritchie and Ken Thompson. Brian Kernighan also played a big role in the history of C and Unix, but never became quite as iconic as the duo that were Richie and Thompson. In the 1980s, the POSIX standard was invented to help standardize Unix systems, as there were many companies who were making their own forks of Unix such as Sun Microsystems (later aquired by Oracle). As you probably know, Linus Torvalds eventually created the Linux kernel, which was heavily based off Unix and is still being maintained by Linus and the Linux community to this very day. The C programming language was inspired by languages like B, PASCAL, and FORTRAN. C has had many releases over the years. The first release was called K&R (Ken and Ritchie). K&R had very sloppy documentation, and only covered what Brian and Dennis felt was necessary to document. K&R also permitted a lot of undefined behaviour which meant that writing certain code on one system might have completely different effects from another. It wasn’t until 1989 that we got ANSI C, also known as C89. The American National Standards Institute (ANSI) created a standard specification of C; something that programmers and compiler engineers could unanimously appeal to. ANSI C still wasn’t perfect, however. In 1990, the International Organization for Standardization (ISO - not to be confused with the file format) created another specification of C known as ISO/IEC 9899:1990, aka. C90. C89 and C90 are essentially the same standard, which is why C90 is usually overlooked. In 1995, the ISO published an extension to C90 which added some new features. This specification was called ISO/IEC 9899/AMD1:1995, or simply, C95. In 2000, ANSI adopted the ISO/IEC 9899:1999 standard aka C99. In 2011, another revision, informally known as C1X was created aka C11. The newest release to date is C23, which again, adds a few keywords and features. The GNU project has their own variants for each of these standardizations to coincide with glibc (which I’ll cover in a bit).

Intro:

The C programming language is what most programmers would call a “low-level” programming language, though at the time it was created, it was actually thought of as a high-level language. By low or high-level, we are refering to the level of abstraction when compared to either assembly or the machine code that C is compiled down into. Most modern languages contain a lot more abstract components and features compared to C, which is why we tend to think of C as being a low-level language these days. This is also the reason that C is typically used to write operating systems and device drivers. In order to communicate with hardware, we must be able to read and write to registers, device drivers, the kernel, etc. These all fundamentally boil down to manipulating memory – something that C abstracts very little from us. I will try not to assume too much of your skill set since C will likely be one of the first languages that you learn. Despite it being a very challenging language to master, I think it is still the best spot to begin learning how to program. The document is formatted as a linear progression starting from basic concepts and working up to more difficult ones. I will also be talking quite a bit about Unix/Linux since C is most easily developed on a Unix-like operating system. In my opinion, Linux is by far the best ecosystem to be writing C, but that is not to say other operating systems are not an option, just that they may pose unique challenges. If you are not very familiar with Unix/Linux, you can skip over some of the system-related aspects of the document and hopefully still walk away understanding the jist of the syntax.

Variables:

Where better to start than variables? Arguably the most basic yet essential units of code are variables. If you’re familiar with programming at all, you know that variables contain values and that the values they contain can assume various types, known as data types. data types always have a specified size in memory. These sizes are defined within the language specification, but can sometimes vary depending on the platform architecture. In other words, the size of an int may vary from a 32-bit machine to a 64-bit machine. Unfortunately, this can get quite confusing. I will be covering each of the primitive data types individually. This might seem like a lot of stuff to remember, however, understanding the size of variables is *very* important when writing C code. We must be much more considerate of how much space we are occupying when writing code in C.

void:

void is a unique data type in the sense that it is essentially the only non-numeric primitive type. The void type is what we call a “unit type”, which is a commonly recurring type found in other programming languages as well. It is both a valid type and an invalid type simultaneously. We cannot create a variable of type void, unless it is a pointer of type void\*, which we will get to later. void is primarily used as the return type for functions. Essentially, it tells the compiler that the function returns nothing, which is why it makes sense for functions, but less so for variables (since a variable cannot store nothing). An empty return statement is valid for functions that return type void, but it is not required (this is typically only used for early returns). As an interesting but useless tidbit, void occupies a single byte of memory, even though it stores no data. This can be observed by printing sizeof(void).

char:

char, short for character, is a *numeric* data type. A char is *always* stored as an integer which is usually translated to a glyph on the screen. You will often hear from folks on the internet that a char occupies 1 byte in C. This is not technically correct. A char in C is at *minimum* 1 byte, and often occupies 1 byte since we typically use ASCII or UTF-8, but in UTF-16, a char is 2 bytes (16 bits = 2 bytes). The actual size of char depends on the locale of your system – something that can be changed from within your C program (see my POSIX programming notes for more info on that). Assuming that a char is 1 byte in most cases, the range can either be -128 to 127 if signed, or 0 to 255 if unsigned. Valid character literals are considered to be any one character surrounded by single quotes e.g., ‘z’. Not all valid character literals must be alpha-numeric. For instance, escape characters are also considered as valid character literals. An escape sequence is simply another way of representing a numeric value (normally represented in decimal, octal, or hexadecimal). Typically, characters which are represented as escape sequences have special meaning or functionality. For example, ‘\n’, ‘\t’, ‘\b’, ‘\0’, etc. are all special characters that may or may not be interpreted to have some special functionality (the C specification says nothing about special characters – it is the OS that defines and implements the behaviour for these characters).

OPTIONAL READING (ADVANCED):

Pertaining to XTerm (the standard terminal emulator for X11), character literals become much more confusing. Technically, we are no longer talking about character literals, but rather, XTerm control sequences. The reason I’m discussing Xterm control sequences is because they closely resemble character literals, and can be stored as character literals, though they are not *technically* character literals. XTerm defines 4 types of control sequences:

**C:** A single (required) character

**Ps:** A single (usually optional) numeric parameter, composed of one or more digits

**Pm:** Multiple numeric parameters, separated by semi-colon. Individual values for these parameters are listed with Ps (see above)

**Pt:** A text parameter composed of printable characters

A standard known as ECMA-48 (aka ISO 6429) specifies two types of **code**. **C0** is a 7-bit code, and **C1** is an 8-bit code. **C0** and **C1** are both subsets of **C**. ECMA-48 does not refer to **C0** or **C1** as characters, since the term character is oft confused to mean “visible glyph”. **C0** can be any decimal from 0 to 31 and also decimals 32 and 127, and **C1** can be any integer from 128 to 159. **C0** control bytes are used for all sorts of purposes such as text layout, transmission and device control ,etc. **C1** control bytes are primarily used for displays and printers. **C1** is the set that is related to ANSI escape sequences and VT100 terminals, and thus, the set that we are most interested in. ECMA-48 processes a control sequence until the sequence is terminated by a terminating byte, or until it finds a byte which does not belong to the sequence. Here are some examples of **C1** control characters:

**ESC D | ASCII char IND | hex 0x84:** Represents an index

**ESC E | ASCII char NEL | hex 0x85:** Represents the next line

**ESC X | ASCII char SOS | hex 0x98:** Represents the start of a string

**ESC [ | ASCII char CSI | hex 0x9d:** Begins a control sequence

These are basic control characters that are used by your terminal unbeknownst to you. The curses and ncurses libraries are the best examples for how these control sequences are used in a practical sense. By selecting control sequences that manipulate the terminal output, we can create TUI applications. Let’s look at a practical example by changing the foreground color of your terminal’s text output. By entering the command printf “\033[91mThis is red text\n\033[0m” into your terminal, you will see it output “This is red text” in red. Let’s break down the control sequence that we just printed. The first character is an escape character, which tells the terminal that we are about to print a character literal. 0 in C indicates an octal number, so 033 is interpreted as octal 33. Looking at an ASCII chart, we see that octal 33 is the ESC (escape) character. As we seen earlier, ESC followed by [ is the **C1** control byte to begin a control sequence (also known as CSI (Control Sequence Introducer) in the ANSI control sequence atlas). If we do a bit more digging into the XTerm spec., we find that the control sequence **CSI Pm m** alters character attributes. Remember that **Pm** can be a list of multiple parameters separated by semi-colons, and that parameters are expressed as **Ps**. The **Ps** values for foreground and background color may differ depending on whether or not your terminal uses 16 colors. Anyhow, we see that when **Ps** = 9 followed by 1, it sets the foreground to Red! We also see that 0 sets it back to normal. Now, say we wanted to create blinking and underlined text with a white background and black foreground. All we have to do is find the correct **Ps** values within the specification. The resulting control sequence, which you can test on your terminal is “\033[4;5;90;107mBlinking, Underlined, FG Black, BG White\n\033[0m”. Do a printf of that on your terminal and be impressed. If you’re wondering how I got the **Ps** values for **Pm**, they can be found in XTerm’s documentation here: [https://invisible-island.net/xterm/ctlseqs/ctlseqs.html#h3-Application-Program-Command-functions](https://invisible-island.net/xterm/ctlseqs/ctlseqs.html" \l "h3-Application-Program-Command-functions). Another resource which I found helpful can be found here: <https://www.aivosto.com/articles/control-characters.html>.

int:

int, short for integer, is probably the most commonly used numeric data type in C. int is another data type that may vary in size depending on platform. Typically, it will be 2 bytes on 32-bit platforms, and 4 bytes on 64-bit platforms. I will be assuming that you are on a 64-bit machine for the remainder of the document, meaning that I will treat ints as 32-bit, but be aware that this is generally not a good assumption to make, especially when writing code that is cross-platform. Without specifying the sign, int is *implicitly* set to be signed. Despite this, ints can be explicitly marked as unsigned. For an int that is signed, the range is -2,147,483,648 to 2,147,483,647 and for unsigned, the range is 0 to 4,294,967,295. Integer literals are commonly represented with prefixes or with suffixes.

**Prefixes**: Prefixes begin at the front of a literal. Ex.

Decimal literal (base 10): Has no prefix

Octal literal (base 8): Begins with a 0 eg. 0777, 0123, 0987

Hex literal (base 16): Begins with ‘0x’ eg. 0xFFFF, 0xA0B2, 0x2B

Binary literal (base 2): Begins with ‘0b’ e.g. 0b00001111, 0b01010101. Please note that binary literals are extremely dependent on the compiler you’re using!!!

**Suffixes:** Suffixes are placed at the end of a literal. These are dependent on the integer type. Since int is technically considered to be the default integer type, it does not have its own dedicated suffix. However, we can still give it the ‘u’ or ‘U’ suffix to specify that it is an unsigned int. For example: unsigned int i = 100u; Note that suffixes are mostly *optional*. Other than a few circumstances, they generally only help to add clarity.

short:

short is short form for “short int” (tongue-twister). A short int is *always* 2 bytes i.e. 16 bits. Note that on 32-bit platforms, short is the same size as an int, which basically makes short useless on 32-bit machines. short can also be signed or unsigned. Again, like int, it is implicitly set to be signed if no sign is given. The range for signed short is -32,768 to 32,767 and for unsigned its 0 to 65,535. Short can either be written as just “short” or “short int”. For example: *short shrt = 10000;* or *short int shrt = 10000;* are both acceptable. Same as int, a suffix of ‘u’ or ‘U’ to specify that it is an unsigned short.

long:

long is short form for “long int”. A long int can be 32-bits (4 bytes) or 64-bits (8 bytes), depending on platform. Same as short int, long can either be written as just “long” or as “long int”. longs can be signed or unsigned. long is implicitly signed if no sign is given. For a signed 64-bit long, the range is -9223372036854775808 to 9223372036854775807 and for unsigned, it is 0 to 18446744073709551615! long has its own, dedicated suffix, which is ‘l’ or ‘L’. This can be used in conjunction with ‘u’ and ‘U’. For example, *unsigned long l = 100000000ul;* is valid.

long long:

long long was primarily designed for 32-bit machines where a long was 32-bit. Since int was also 32-bit, long was pretty much useless. Thus, we were given long long, or long long int, which is always 8 bytes. The range will still be the same as what I specified in the previous section, since I was talking about 64-bit longs. long long also has its own suffix ‘ll’ or ‘LL’ which can also be used with ‘u’ or ‘U’. For example: *signed long long int bignum = -999999999999LL;*

float:

We are now moving away from integer types, and entering decimal types, a.k.a. floating-point integers. While still numeric, floating-point types do not represent themselves the same way that normal binary integers do. Instead, they are represented by a format called IEEE-754 by the ISO standard. Note that IEEE-754 is not the only method of representing decimal types (IEEE-754 is known as floating point notation, but there also exists something called fixed point notation, which we will not be covering here, but that I recommend you look up). float is short for “single-precision floating point integer”. This is a fancy way of saying that it’s a 32-bit decimal number. floats *must* be signed because the Most Significant Bit (MSB) is dedicated for storing the sign within their binary format according to the IEEE-754 specification. Therefore, an “unsigned float” would be a compile-time error. floats also have their own suffix which is ‘f’ or ‘F’. For example: float fpl = 57.998779f; The suffix for float is the only *non-optional* suffix. If you omit the suffix for float, it will be promoted to a double, which will occupy twice the space.

double:

doubles are also considered to be floating-point integers, however, double stands for “double precision” since there are twice as many bits in a double as there are in a float, (and therefore double the precision). As I mentioned, floating-point integers cannot be unsigned, and that still applies to doubles. doubles are 64-bits. Unlike float, double does not have any suffix to denote that it is a double. Akin to int, double is assumed if no suffix is given, even if using the keyword “float”. Not much more is to be said here... There is some debate as to whether double is better than float. In my humble opinion, float is sufficient for the majority of calculations, making it preferred over double for programs that require a lot of maths.

long double:

The final primitive type that we’ll be discussing is the long double. long doubles can vary in length, but on a 64-bit machine, they are typically a whopping 128 bits (16 bytes!). long doubles share the same suffix as long, being ‘l’ or ‘L’. Once again, only signed is allowed. On a final note, I don’t recommend using long double unless absolutely necessary. It may not seem like it but allocating 16 bytes a few times can really add up, especially on systems with only a few MB of memory if you’re on an embedded system.

Typedef:

If you are familiar with C, it may seem an odd choice to jump straight into typedefs, but I think it will be important to explain them now in order to help us understand what’s happening later on. A typedef is simply an alias for an existing data type. “typedef” is a keyword in C, which we can use to help make our code shorter or easier to read. For example, take the following line: typedef unsigned long long int ull\_t; Note that the last item in the sequence is the typedef name/alias for the type preceding it. For example, in this case, ull\_t is a typedef for “unsigned long long int”, or in other words, an alias for “unsigned long long int”. Typedefs may seem a bit redundant, since you can always just enter the actual data type, but they are a very common practice in C, which is why I want to explain them now. They can be declared almost anywhere, and conventionally, they end with “\_t” to indicate typedef.

Useful Typedefs:

I think it would be very beneficial to go over some pre-existing typedefs since these will come up a good amount throughout the duration of this document, as well as the duration of your C coding career. I will go over these typedefs the same way that I went over the variables beforehand, albeit in less detail. Keep in mind that these are just aliases.

bool (stdbool.h):

bool is short for boolean. Note that there is no built-in data type for booleans in C. bool was added in C99 under stdbool.h (we haven’t gotten to header files yet, so don’t worry if you don’t know what I’m talking about). bool is actually not a typedef (hehe), it is technically a macro, but it’s a similar idea. You can tell it’s not a typedef since it does not end with the “\_t” suffix. Although booleans only require 1 bit, C does not have any built-in types that are that small, thus bool occupies one byte. stdbool.h also adds definitions for true and false. true is equivalent to 0x01 and false is equivalent to 0x00. Using bool is a good option when you are doing heavy boolean logic and want to add clarity to your code, or when you want to return true or false from a function.

Platform Agnostic Types (stdint.h):

There are a few typedefs which allow you to create integer type variables with the exact number of bits requested, independent of platform. The stdint.h header file consists of the typedefs uint8\_t, uint16\_t, uint32\_t, and uint64\_t, as well as their signed int variants. The ‘u’ in uint stands for unsigned int as you may have guessed. As I mentioned, the very nice thing about these typedefs is that they guarantee the number of bits that you specify on any platform. So, for example, uint16\_t will always allocate 16 bits, no matter what architecture. Here are the typedef definitions for each if you are curious:

typedef unsigned char uint8\_t;

typedef unsigned short int uint16\_t;

#ifndef \_\_uint32\_t\_defined

typedef unsigned int uint32\_t;

#define \_\_uint32\_t\_defined

#endif

#if \_\_WORDSIZE == 64

typedef unsigned long int uint64\_t;

#else

\_\_extension\_\_

typedef unsigned long long int uint64\_t;

#endif

Don’t worry if you don’t understand what some of the lines are doing. The important part is just that you understand that these guarantee the number of bits that they claim.

size\_t and ssize\_t (stddef.h and sys/types.h):

size\_t and ssize\_t are very commonly used typedefs. The number of bits that it occupies will always be equal to the architecture size. For example, on 32-bit hardware, (s)size\_t would 32 bits, but on a 64-bit machine, (s)size\_t would be 64 bits. size\_t is the unsigned variant, whereas ssize\_t is the signed variant (the first ‘s’ stands for signed). The reason people like this typedef is primarily readability. If you’re storing the number of bytes something occupies, or perhaps the number of items an array can hold, (s)size\_t is more legible than using something like int. What’s perhaps a bit confusing is the header files that these typedefs are defined in. size\_t is defined in stddef.h, but since ssize\_t is for some reason defined in sys/types.h. sys/types.h happens to include stddef.h, and stdio.h happens to include sys/types.h, so in order to access both of these, we just have to include stdio.h.

wchar\_t (stddef.h):

wchar\_t, which stands for “wide character” is an alternative to char. wchar\_t will have a size in bytes large enough to accomodate the biggest character set within the set of supported locales. Naturally, the standard printf() function was never designed for unicode, so there’s a bit more work that needs to be done in order to print unicode characters to the screen. First, we need to set a locale via setlocale(), defined in locale.h. This function accepts a category and a locale. The category specifies what will be affected, and the locale specifies the character encoding that will be used. We also need to use the wprintf() function defined in wchar.h rather than printf(). The code looks something like the following:

#include <wchar.h>

#include <locale.h>

int main(void) {

wchar\_t emoji = 0x0001F600;

setlocale(LC\_CTYPE, "");

wprintf(L"%lc\n", emoji);

return 0;

}

LC\_CTYPE specifies that we want to set the locale only for regular expressions, character classification, character conversion functions, and wide-character functions. The empty string indicates that we want to receive a locale that makes sense according to our environment. In my case, it will likely resort to using en\_US.UTF-8. Then we simply print the wchar using wprintf(). Note the ‘L’ prefix on the string literal. This indicates that each character in the string literal should be treated as a wchar\_t.

Note:

For all of these typedefs (and any function definitions too), you can search the man pages for their header files to get more information. For example, to find out more information about bool, you can enter man stdbool.h and it will give a list of things that it defines.

Preprocessor Directives:

We are now going to spend a long time talking about preprocessor directives, which will probably confuse you, but that’s okay. We haven’t delved deep into how the compiler works at this point, and I’d like to keep it like that for now, but there is one thing that I must quickly explain in layman’s terms. The compiler has many stages in its pipeline, and the first of those stages is known as pre-processing. Pre-processing happens before the actual compilation of the program. In essence, the pre-processor will parse the file for special statements called pre-processor directives. Pre-processor directives start with the hash symbol ‘#’ (the correct name for the hash or pound symbol is an octothorp)! These lines will be processed as commands prior to all other lines in the source file. Unlike regular code statements, you should not end pre-processor directives with a semi-colon unless you know what you’re doing. Here are a few common pre-processor directives:

**#define:** Substitutes a preprocessor macro.

**#undef:** Undefine a preprocessor macro.

**#include:** Inserts a particular header file into the current file.

**#ifdef:** Returns true if macro is defined.

**#ifndef:** Returns true if macro is undefined.

**#if:** Tests is a compile condition is true.

**#elif:** Equivallent to else if.

**#else:** The default branching condition.

**#endif:** Ends preprocessor #if statement.

**#error:** Prints error message on stderr.

**#pragma:** Issues special commands to the compiler, using a standardized method.

Using these directives is an art that is difficult to master. Let’s start going through them one by one. I will be grouping some directives together such as #if, #elif, and #else for example.

#define:

Aside from #include, #define is the directive that you will see used most and will be most beneficial to you. #define will create what’s known as a macro, which is sort of like a label that expands to whatever text you provide it. Macros don’t take up any physical address space in RAM because they are not actually variables that get initialized anywhere. The #define directive takes in two “parameters”: the name of the macro, and the macro’s assigned value. For example, we might be interfacing with a device where the I/O has been memory mapped to a specific address in RAM. In order to read/write to/from this port, we can define the base address like so:

#define PORTA 0x2000

In this case, whenever C comes across the macro PORTA, it is *substituted* with the literal text “0x2000”. #define can be dangerous because we can do things that are not permitted/defined within the C specification. For example, we can define closing and opening curly braces (something you should never do)!!!

#define OPEN\_BRACE {

#define CLOSE\_BRACE }

if (true) OPEN\_BRACE

// Do something

CLOSE\_BRACE

Notice that the standard for macros is to make them capitalized. While it is not required to capitalize macros, it is still heavily encouraged for the purposes of readability and avoiding namespace clashing.

#include:

#include is the most commonly used preprocessor directive, since it is required within nearly every source file that you write. Again, we haven’t discussed header files yet, but let’s just say that they are essentially files which contain definitions about certain variables and functions. The #include directive is used to include these files into our code. #include takes one “argument”, which is the .h file that we are including. There are two options for how we include them. One is to use triangle brackets (<, >) and the other is to use double quotes (“ ”). Triangle brackets indicate that the header file can be found in a location on the host system that belongs to the compilers lookup path. On linux, the linker first searches the $LD\_LIBRARY\_PATH environment variable, followed by the $PATH variable (see the man page for ld to get a more accurate list of directories that the linker searches). Triangle brackets are typically used for header files that are already installed on your system due to being commonly used by other programs (e.g., header files that belong to the C runtime library). The double quotes indicate that you’d like to manually enter the path. This path can be a full path starting from your root directory, otherwise, the program will assume a relative path starting from the directory of the .c file in which #include was used. Ex.

/\* Common header file that belongs to libc. Probably exists under /usr/shared/lib \*/

#include <stdio.h>

/\* Relative path. Header file likely belongs to your project \*/

#include “../../header.h”

One final note about the #include directive is that this does not “import” code in the same manner as a language like Java. The contents of header files that are included with #include get substituted in-place i.e. at the location of the #include directive. In other words, #include essentially copies all the code in the header file and pastes it into the source file. Considering that #include is a preprocessor directive, this means that no semantic analysis takes place when the file is inserted. In other words, C will not complain if we include any file that exists on the system until we actually try to compile. I’ve seen certain people (though this is bad practice) #include other C source files directly. This makes #include a very powerful command indeed. We will certainly review this directive once we’ve covered header files, but hopefully that gives you a general idea of how it works.

#undef:

#undef, as you may have guessed, undefines a macro that was previously defined. The use cases for #undef are a bit niche, but there are certainly appropriate times to use it. To put it in other words, #undef destroys a macro. The danger with #undef is that it allows us to redefine a macro. This might sound like a nice thing, but it only leads to confusion and should be avoided at all costs! For example:

// Don’t do the following!

#define A\_MACRO 0

#undef A\_MACRO

... // Some code

#define A\_MACRO 1

If you genuinely intend to undefine the macro, and don’t plan on redefining it again, then #undef is perfectly fine to use. And perhaps, if you are very cautious, I would even go as far as to say that it’s okay to #define, #undef, and #define again, so long as the second #define sets the macro back to its original value when it was first #define(d) (but generally, just avoid this where possible).

#ifdef, #ifndef:

#ifdef and #ifndef are really nice pre-processor directives because they allow us to check if a macro is or isn’t defined. In conjunction with #define and #undef, this can become very useful. This is commonly used to execute sections of code based on the computer’s architecture or based on certain build options. We will be going over something called include guards (which use these directives), as well as a few other pre-processor idioms after I’m finished explaining all the directives. The only thing that you should know for now is that #ifdef returns true if the macro provided is defined, and #ifndef returns true if the macro provided is *not* defined.

#if, #elif, #else, #endif:

These 4 directives are equivalent to if, else if, and else. No matter how many #ifs or #elifs are used, an #if directive must always close with #endif. Once again, this is useful for blocking and controlling certain segments of code. We will also be going over examples in a bit.

#error:

The moment that #error is reached, the program is aborted, and the error message supplied after abort is printed to stderr. This is probably a good time to talk about how we can have multi-line pre-processor directives. Since newline characters normally signify the end of a preprocessor directive, we must use the backslash ‘\’ to indicate that we want to continue the next line. For example:

#error This is a very long error message\

that i am printing to stderr

Note that the backslash allows us to continue the preprocessor directive onto the next line. This can aid with readability if you have a very long argument. Something *very* important to note is that adding an additional space after the backslash will cause the newline to remain, which can lead to code that is extremely hard to debug! For this reason, I recommend that you have your text editor remove trailing whitespace. Most editors have a setting to enable this feature, or if you are using something like Vim, there are autocommands which can remove trailing whitespace on write.

#pragma:

#pragma might just have to be a whole guide on its own, but it’s okay because it’s not used much in C. #pragma actually has a bunch of subcommands called “pragma directives” that do extra work within the pre-processor. Pre-processor directives are not always enough to do everything that we want, so pragma directives can grant additional control. #pragma is most commonly used to supress certain warnings made by the compiler. Here is a list that you can find on Microsofts C++ documentation <https://docs.microsoft.com/en-us/cpp/preprocessor/pragma-directives-and-the-pragma-keyword?view=msvc-170>:

[alloc\_text](https://docs.microsoft.com/en-us/cpp/preprocessor/alloc-text?view=msvc-170) [auto\_inline](https://docs.microsoft.com/en-us/cpp/preprocessor/auto-inline?view=msvc-170) [bss\_seg](https://docs.microsoft.com/en-us/cpp/preprocessor/bss-seg?view=msvc-170) [check\_stack](https://docs.microsoft.com/en-us/cpp/preprocessor/check-stack?view=msvc-170)  
[code\_seg](https://docs.microsoft.com/en-us/cpp/preprocessor/code-seg?view=msvc-170) [comment](https://docs.microsoft.com/en-us/cpp/preprocessor/comment-c-cpp?view=msvc-170) [component](https://docs.microsoft.com/en-us/cpp/preprocessor/component?view=msvc-170) [conform](https://docs.microsoft.com/en-us/cpp/preprocessor/conform?view=msvc-170)  
[const\_seg](https://docs.microsoft.com/en-us/cpp/preprocessor/const-seg?view=msvc-170) [data\_seg](https://docs.microsoft.com/en-us/cpp/preprocessor/data-seg?view=msvc-170) [deprecated](https://docs.microsoft.com/en-us/cpp/preprocessor/deprecated-c-cpp?view=msvc-170) [detect\_mismatch](https://docs.microsoft.com/en-us/cpp/preprocessor/detect-mismatch?view=msvc-170)  
[endregion](https://docs.microsoft.com/en-us/cpp/preprocessor/region-endregion?view=msvc-170) [fenv\_access](https://docs.microsoft.com/en-us/cpp/preprocessor/fenv-access?view=msvc-170) [float\_control](https://docs.microsoft.com/en-us/cpp/preprocessor/float-control?view=msvc-170) [fp\_contract](https://docs.microsoft.com/en-us/cpp/preprocessor/fp-contract?view=msvc-170)  
[function](https://docs.microsoft.com/en-us/cpp/preprocessor/function-c-cpp?view=msvc-170) [hdrstop](https://docs.microsoft.com/en-us/cpp/preprocessor/hdrstop?view=msvc-170) [include\_alias](https://docs.microsoft.com/en-us/cpp/preprocessor/include-alias?view=msvc-170) [init\_seg](https://docs.microsoft.com/en-us/cpp/preprocessor/init-seg?view=msvc-170)  
[inline\_depth](https://docs.microsoft.com/en-us/cpp/preprocessor/inline-depth?view=msvc-170) [inline\_recursion](https://docs.microsoft.com/en-us/cpp/preprocessor/inline-recursion?view=msvc-170) [intrinsic](https://docs.microsoft.com/en-us/cpp/preprocessor/intrinsic?view=msvc-170) [loop](https://docs.microsoft.com/en-us/cpp/preprocessor/loop?view=msvc-170)  
[make\_public](https://docs.microsoft.com/en-us/cpp/preprocessor/make-public?view=msvc-170) [managed](https://docs.microsoft.com/en-us/cpp/preprocessor/managed-unmanaged?view=msvc-170) [message](https://docs.microsoft.com/en-us/cpp/preprocessor/message?view=msvc-170) [omp](https://docs.microsoft.com/en-us/cpp/preprocessor/omp?view=msvc-170)  
[once](https://docs.microsoft.com/en-us/cpp/preprocessor/once?view=msvc-170) [optimize](https://docs.microsoft.com/en-us/cpp/preprocessor/optimize?view=msvc-170) [pack](https://docs.microsoft.com/en-us/cpp/preprocessor/pack?view=msvc-170) [pointers\_to\_members](https://docs.microsoft.com/en-us/cpp/preprocessor/pointers-to-members?view=msvc-170)  
[pop\_macro](https://docs.microsoft.com/en-us/cpp/preprocessor/pop-macro?view=msvc-170) [push\_macro](https://docs.microsoft.com/en-us/cpp/preprocessor/push-macro?view=msvc-170) [region](https://docs.microsoft.com/en-us/cpp/preprocessor/region-endregion?view=msvc-170) [runtime\_checks](https://docs.microsoft.com/en-us/cpp/preprocessor/runtime-checks?view=msvc-170)  
[section](https://docs.microsoft.com/en-us/cpp/preprocessor/section?view=msvc-170) [setlocale](https://docs.microsoft.com/en-us/cpp/preprocessor/setlocale?view=msvc-170) [strict\_gs\_check](https://docs.microsoft.com/en-us/cpp/preprocessor/strict-gs-check?view=msvc-170) [system\_header](https://docs.microsoft.com/en-us/cpp/preprocessor/system-header-pragma?view=msvc-170)  
[unmanaged](https://docs.microsoft.com/en-us/cpp/preprocessor/managed-unmanaged?view=msvc-170) [vtordisp](https://docs.microsoft.com/en-us/cpp/preprocessor/vtordisp?view=msvc-170) [warning](https://docs.microsoft.com/en-us/cpp/preprocessor/warning?view=msvc-170)

Predefined Macros (stdio.h):

Before we jump into some use-cases for pre-processor directives, I wanted to briefly touch on some pre-defined macros that are available to us. Most of these are included within stdio.h so you will need to include that in order to have access to them.

**\_\_DATE\_\_:** Expands to the current date in the format MM DD YYYY.

**\_\_TIME\_\_:** Expands to the current time in the format HH:MM:SS.

**\_\_FILE\_\_:** Expands to the name of the file that encloses it.

**\_\_LINE\_\_:** Expands to the line number that the macro was declared on.

**\_\_FUNCTION\_\_:** Prints the function name of the enclosing scope.

**\_\_func\_\_:** Effectively replaces \_\_FUNCTION\_\_ as of C99. This macro expands to *static const char \_\_func\_\_[] = “function\_name”;*. It is therefore, treated as a variable, rather than a string literal.

**\_\_STDC\_\_:** Defined as 1 when the compiler complies with the ANSI standard (C89/C90). Note that there are also compiler options that check for this at compile time e.g., the -ansi and -pedantic flags.

**\_\_STDC\_VERSION\_\_:** Prints a timestamp in the format YYYY MM that describes the C standard that the compiler is using.

**\_\_STDC\_HOSTED\_\_:** Defined as 1 when the compiler’s target is a hosted environment. A hosted environment is an environment which implements the full libc standard.

**\_\_cplusplus:** Similar to \_\_STDC\_VERSION\_\_, \_\_cplusplus also expands to a version string in the same format. It is used to test if a header was compiled by a C compiler or a C++ compiler.

**\_\_OBJC\_\_:** Defined as 1 when the Objective-C compiler is in use (Objective-C was, once upon a time, used by Apple).

**\_\_ASSEMBLER\_\_:** Defined as 1 when preprocessing assembly language.

**\_\_VA\_ARGS\_\_ :** List of variadic arguments (will cover in more depth in a later section).

Note that most of these pre-defined macros are compiler intrinsics/built-ins defined by GNU, and therefore, may not exist if you are not compiling with GCC. LLVM’s Clang, and Mircosofts MSVC will have their own pre-defined macros as well. More information can be found here: <https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html>.

Reserved Identifiers:

As we just seen, a lot of the pre-defined macros begin with two underscores. The common convension is that a single underscore is reserved for low level system variables and two underscores are reserved for compiler intrinsics/built-ins (more on those in a later section).

Common Macro Use-Cases/Idioms:

There are a couple of recurring patterns that are repeated in programming, usually for good reason. These patterns are called idioms. Macros have a couple of commonly used idioms that are useful for specific use-cases. The following couple of sections will cover each of the most common and useful macro idioms.

Include Guard Idiom:

Anytime that we use the #include directive, the directive will copy and paste the contents of the provided header file into the source file a.k.a. the .c file. If we have multiple .c files, it is possible that multiple of them might include the same header files. This is a waste of energy on the compilers part, and worse yet, leads to re-definition errors. In order to make sure that a header file only gets included once, we often use the include guard idiom. The include guard is added within the header file and prevents the header from being #include(d) more than once. Here is what the include guard looks like:

#ifndef FOO\_H

#define FOO\_H

// All of the header code/definitions

#endif

What’s happening here is that the #ifndef checks to see if FOO\_H is a macro that has previously been defined. If it has been defined, then the #ifndef returns false, and the entire include guard is skipped, otherwise, we #define FOO\_H (we don’t need to give it any value, it just needs to exist), then we insert all the code and/or definitions that would normally be added to our header file. Keep in mind that the entire header gets cut and pasted where we #include the header file, which would include our include guard. The next time that a .c file tries to #include foo.h, the #ifndef returns false since we #define(d) FOO\_H previously, and therefore, we skip to the #endif statement at the bottom of the file, and foo.h does not get included a second time. This all sounds very confusing if it’s your first time coding with pre-processor directives, but it will become second nature to you very quickly.

Disabling a Region of Code:

Rather than commenting out a region of code using a multi-line comment, we can easily disable a region of code by using the #if and #endif macros. It’s as simple as follows:

#if 0

// Code to disable

#endif

This simple trick can comment out code the same as a regular multi-line comment. The #if 0 always evaluates to false, since 0 is evaluated as false in boolean expressions in C. You may want to do this to enable or disable a region of code. Whilst you could just use a multi-line comment, it may get lost amongst the other comments, making it more difficult to read.

#define in #include:

This idiom is somewhat confusing upon first glance, but I’ll do my best to explain. I haven’t actually covered this yet, but it is possible to #define an expression as a macro. Typically, these are small functions which are often single line (sometimes called inline) expressions or are just very simplistic. Arguments can be passed into macros using the round brackets () similar to methods/functions. This will be taken advantage of in the following idiom:

#define PROJ\_DIR /home/neil/devel/proj

#define INCLUDE\_DIR(h\_file) “PROJ\_DIR/include/h\_file”

#include INCLUDE\_DIR(bar.h)

Here we #define two macros: a project directory, and a macro for the project’s include directory where the .h files will be located. On the third line, we include the macro INCLUDE\_DIR and pass in bar.h (the header that we want to include) as a parameter. In this case, the second line with the macro INCLUDE\_DIR is almost acting more like a function in a way. It is defined as “PROJ\_DIR/include/h\_file”. PROJ\_DIR gets substituted since it is a macro defined on the first line, so INCLUDE\_DIR actually gets expanded to “/home/neil/devel/proj/h\_file”. But we’re not done yet because we pass in bar.h as an argument on the following line. The parameter to that argument is called h\_file, so all instances of h\_file get substituted with bar.h. This means that INCLUDE\_DIR actually gets substituted as “/home/neil/devel/include/bar.h”. Therefore, the #include will be substituted as such: #include “/home/neil/devel/proj/include/bar.h”

Cheap Debugging:

In beginner code, it is not uncommon to see a million print statements that print out “made it to line 5” or something to that effect. If we are truly too lazy to use an actual debugger, then there is a slightly better way to do this (although you should probably learn how to use gdb). We haven’t covered printf() yet, but it is C’s function for printing text to stdout and works the same as System.out.format() in Java.

#define HERE printf(“file: %s, function: %s, line: %d\n”, \

\_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_)

HERE;

// blah blah

HERE;

// blah blah

Simply by adding the macro HERE, we print out the function name and line number that we reached. Note that we have a semi-colon after HERE. This is because the statement printf(“%s:%d\n”, \_\_FUNCTION\_\_, \_\_LINE\_\_) must end in a semi-colon. Since it’s bad practice to put semi-colons in macros, we place it after the macro instead.

Pre-processor Function Macros:

I noted before that it is possible to create small functions using macros. I emphasize the word “small” because it is generally frowned upon to use macros for defining entire functions. Since macros are substituted prior to compilation, it is very common to see operations occur out of order. For example, take the following macro:

#define SECS\_TO\_MSECS(s) s \* 1000

some\_timer\_function(SECS\_TO\_MSECS(5 + 5)); // Trying to pass in 10, 000 to some\_timer\_function

While it seems like this should work (converting 10 seconds to 10,000 milliseconds), we actually end up with 5005ms. This is because the macro is expanded before being compiled. The parameter ‘s’ gets substituted as 5 + 5, thus SECS\_TO\_MSECS is evaluated as 5 + 5 \* 1000. Order of operations states that the multiplication should occur prior to the addition, so this evaluates to 5005.

The common solution to this issue it to surround everything in brackets when you pass in parameters to a macro. The #define for SECS\_TO\_MSECS should be re-written as follows:

#define SECS\_TO\_MSECS(s) ((s) \* 1000)

This would expand ‘s’ as (5 + 5) so that the addition would occur before the multiplication.

do while(0) Idiom:

The do while(0) idiom is sort of a confusing one because we must first understand the problem that it solves. Let’s say that we want to have a macro that runs 2 or more functions in a row. You might consider writing something like the following:

#define RUN\_FUNCS(x) do\_this((x)); \

do\_that((x))

if(condition)

RUN\_FUNCS(5);

The if condition would expand to:

if(condition)

do\_this(5);

do\_that(5);

Assuming that you were trying to fit both functions into the if statement, this will not result in the desired behaviour. If no scope is provided with the curly braces in an if statement, only the line that comes directly after the if will be evaluated. In this case, do\_this(5) would be evaluated if the if condition was true, and then, since do\_that(5) is not considered to be inside the if statement, it will get ran no matter what. If the if condition is false, do\_this(5) fails to run, but do\_that(5) still runs.

In order to have both function calls be within the if statement block, we can use the do while idiom. This looks like the following:

#define RUN\_FUNCS(x) do { \

do\_this((x)); \

do\_that((x)); \

} while(0)

Remember that 0 evaluates to false in boolean expression in C, so this do-while loop will only run once since while(0) will return false and exit the loop. However, because do\_this() and do\_that are both enclosed within the do-while loop, the if statement will now include both functions.

if(condition)

RUN\_FUNCS(5);

Expands to:

if(condition)

do {

do\_this(5);

to\_that(5);

} while(0);

Which for all intents and purposes is identical to:

if(condition) {

do\_this(5);

do\_that(5);

}

Enums:

Wasn’t quite sure where to put this section, but it sort of fits in with data types, typedefs, and macros, so I’ve decided to paste it here. If you’re not familiar with what enums are, don’t worry, they are quite simple. Understanding enums will make you a much better programmer, even if they might come across as pointless upon first glance. Essentially, enums are just a list of variable names that have some start value, and increment in value as the list grows. Here is what the declaration of an enum looks like:

enum Errors {

BADBLOCK,

NOMEM,

CRASH,

UNKNOWN

};

As you can see, I’ve created a list of variables which each have some name pertaining to an error that might arise in my code. Enums implicitly have a start value of 0, so BADBLOCK would be = 0, NOMEM would be = 1, and so forth. However, we can change that start value with the assignment operator like so:

enum Errors {

BADBLOCK = 1,

NOMEM,

CRASH = 5,

UNKNOWN

};

Now, BADBLOCK will be = 1, NOMEM will be = 2 (since enums are incremental), CRASH will be 5, and UNKNOWN will be 6. Enums must be of type int. You may think of enums as a list of correlated integer values. Enums are often used for setting flags or defining correlated states.

Note that it is not uncommon to use enums for other purposes, such as defining a list of correlated characters. Since characters are numeric types in C, it is perfectly acceptable to use enums as aliases for character literals. For example, say that we want to define read, write, and execute modes. We could store those as an enum like so:

enum Permissions {

READ = ‘r’,

WRITE = ‘w’,

EXEC = ‘x’

};

The benefit of enums is primarily readability. An enum provides a nice block of categorized labels. It is also helpful when reading a function signature if the parameter is of type Permissions (for example), rather than just int. Another potential benefit is that smart editors will usually detect when you’ve failed to handle all labels for an enum within a switch statement, among other things.

Declaration vs. Definition and Implicit vs. Explicit:

I would like to cover some terminology to aid you as I continue discussing the theory. The terms **declaration** and **definition** get thrown around a lot, and you may come to the false conclusion that these can be used interchangeably. The term **declaration** means that we declare the existence of something to the compiler. The same goes for functions in C. We first must declare a function’s prototype, and then **define** the contents of the function. Here are examples of declaring and defining a variable and a function:

int a; /\* Variable declaration \*/

int a = 10; /\* Variable definition \*/

int func(int a); /\* Function declaration \*/

int func(int a) { /\* Function definition \*/

a++;

return a;

}

Another thing that will get thrown around a lot for the remainder of your programming career are the terms **implicit** and **explicit**. Implicit means that it is “not written in bold” i.e. it’s not entirely obvious. Explicit means that you have made your intentions abundantly clear. A common example of implicit and explicit come up when we are casting one type to another. Consider the following: double d = 65; Here we have provided the variable d (of type double) with an integer literal. It is **implicit** that 65 is converted to a double by the compiler. In other words, 65 becomes 65.0, and is stored according to IEEE-754 whether you realize it or not. In order to make this **explicit**, we would use the cast operator like so: double d = (double)65; In this case we’ve explicitly stated that 65 is being cast to a double. Either is fine, but explicit statements often help with legibility and display intent to the reader.

Functions:

I think it’s now appropriate to talk about functions a little bit. Functions are declared/defined similarly to most procedural languages. They have a return type, a name, and an optional parameter list. These three things make up the function’s “signature”. In C, we are not able to override functions (technically speaking). A function “prototype” is synonymous with function “declaration”. As I mentioned, we must declare a function’s prototype before the function can be defined. Ignoring a function’s prototype can lead to undefined behaviour (and some very nasty bugs). This is the primary use for header files – they contain all of the function prototypes to avoid clutter in the source file. You do not *have* to put prototypes within a header file, but for larger code it becomes practical in order to maintain readability. Note that main() is a special function which the compiler expects to find once and only once. There are two prototypes of main that are defined by the POSIX standard:

int main(void);

int main(int argc, char \*argv[]);

Note that argv is sometimes also written as char \*\*argv (it makes no difference, so long as we can input an array of strings). Other variations of main are not POSIX compliant. For example, a common mistake for beginners is to define main as: int main() {} without the void keyword, however, this is technically incorrect, even though your program will still compile and execute. The second prototype of main is only useful if you want to take in command line arguments. argc (argument count) is the number of arguments, and argv (argument vector) is an array of strings. The first argc indexes of argv will contain each parameter passed to the program (separated via whitespace). For example, if I run an executable called runme with the command line arguments hello and world like so: ./runme hello world, then argc will be 3 (runme is considered to be the 0th argument) and argv will contain the strings “./test”, “hello”, and “world” in indexes 0, 1, and 2 respectively.

Now let’s say that you wanted to create a basic hello world program. We will create a function called say\_hello(). The prototype for say\_hello() will be declared in a header file called hello.h, and the definition for say\_hello() will be defined in a source file called hello.c. Let’s look at how that might look:

/\* hello.h \*/

#ifndef HELLO\_H

#define HELLO\_H

void say\_hello(void);

#endif

/\* hello.c \*/

#include <stdio.h>

#include “hello.h”

void say\_hello(void) {

printf(“Hello World!\n”);

}

int main(void) {

say\_hello();

return 0;

}

And now we’ve written our first hello world program! + an include guard to prevent the code from accidentally being included in more than one source files. We declared the function prototype for say\_hello() which has a return type of void i.e. it returns nothing, and a parameter of type void i.e. it takes in nothing. In the .c file, we #include <stdio.h> and “hello.h” (we use double quotes assuming that hello.h and hello.c are placed in the same directory). Then we define say\_hello(). We do this by giving it a scope with the curly braces rather than ending it with a semi-colon. We call a function called printf() which is included in stdio.h and finally, we add our main function which calls say\_hello() and returns with an exit status of 0 (which indicates that the program completed without any errors).   
  
  
Note: If you ever have to look through a very old C codebase it is possible that you might stumble upon K&R-style declarations for functions. Prior to the ANSII reformed standard, people used to declare their functions something like this:

void foo(a, b, c)

int a;

short b;

char \*c;

{

...

}

This style of declaration has since been deprecated, but I actually did run across this by chance during one of my first programming jobs and was not familiar with it, so in the rare case that you ever see something like above, you’ll now know why.

getopt(): getopt(int argc, char \*const argv[], const char \*optstring)

The getopt() function is a very useful function for parsing command line options entered in the terminal. In order to use this function, you must #include <unistd.h>. It takes in three parameters. The first parameter is the number of arguments, and the second parameter is an array of strings. Sound familiar? These two arguments are meant to match the ones from main(int argc, char \*argv[]). The third parameter is a string that you provide to specify what options should be available. This function can be really confusing when it comes to return values, but I’ll attempt to explain it. unistd.h defines certain global/external variables. These extern variables include char \*optarg, which is a string, optopt, which is a char, and optind, an int. Depending on the return value of getopt(), these variables may or may not be set. Here is a list of possible return values for getopt():

- Returns -1 if there are no more options to be parsed

- Returns ‘?’ if getopt() comes across and unrecognized option or if a value is not provided after an option flag when one was expected. Unrecognized options are stored in optopt.

- If we provide a ‘:’ before the option in optstring, instead of returning ‘?’ when an expected value is missing, getopt() returns ‘:’.

I recommend that you read the official man pages on this function if you are still confused, as it is definitely the best resource for this particular function. Here is a brief example of how getopt() can be used:

int main (int argc, char \*argv[]) {

int opt;

/\* Provide ‘:’ before each option that expects a value to proceed it \*/

while((opt = getopt(argc, argv, ":f:d:s")) != -1) { /\* f = file, d = destination, s = source \*/

switch (opt) {

case 'f':

printf("User provided file: %s\n", optarg);

break;

case 'd':

printf("User provided destination location: %s\n", optarg);

break;

case 's':

printf("User provided source location: %s\n", optarg);

break;

case '?':

printf("Unrecognized option: %c\n", optopt);

break;

case ':':

printf("Option expects a value to proceed it\n");

break;

default:

printf("Should not be possible to reach here\n");

break;

}

}

/\* optind stores the index of the previously parsed argument and is useful

for collecting additional arguments that were not parsed by getopt() \*/

while (optind < argv) {

printf(“Extra arguments: %s\n”, argv[optind]);

optind++;

}

return 0;

}

The Compilation Process:

I feel that here, we reach a fork in the road when it comes to me trying to explain the way our code works. It is time that we understand the compilation process. Unfortunately, the compilation process is quite messy and difficult to understand, but we will have to trek across this swamp before we can continue walking on dry land. As you’ve learned, the compiler has at least two stages: the pre-processing stage and the actual compilation stage. Depending on who you ask though, you will get varying responses about how many stages of the compiler there really are. I like to say that there are really 5 stages in the compilation pipeline. These 5 stages go as follows: Pre-processing, compilation, assembly, linking, loading. The compiler does not typically do all of these things by itself. It may rely on other programs that only focus on one specific section of the pipeline. For example, on Unix, we had tools such as **cc** (the c compiler), **as** (the assembler), **ar** (the archiver), and **ld** (the linker/loader). This sequence of programs is typically referred to as the “C Compiler Toolchain” since each tool is used sequentially, one after the other (like links in a chain). I will try to explain each step of the compilation process, but first, let me explain some basic things.

A .c file is often called a “source file” because it contains most of our source code. A header file (sometimes called an include file), or .h file, often contains declarations of certain functions or variables that we need to include in our source files (note that header files are still considered to contain source code, even if they aren’t called “source files”). A source file, along with all of the header files that it includes is called a “translation unit” or “compilation unit”. When a translation unit is compiled, it outputs an object file (.o). Additionally, you should know that we can create an archive (a folder) of .o files called a static library, which ends with the .a extension. On Windows, a static library would have the .lib extension. We can also create shared libraries called .so files on Linux (which are equivalent to .dll files on Windows). Finally, you should know what an executable file is. Executables can link with static libraries, and/or link with shared objects. An executable which contains only static libraries is called a static executable, and an executable that loads one or more shared objects at runtime is called a dynamic executable. With those terms out of the way, I’ll explain how the compilation pipeline works one step at a time:

Pre-processor:

The pre-processor phase happens before the compilation phase, as you know. The pre-processor primarily deals with pre-processor directives. That means substituting macros, expanding include files, and stripping comments. It is also in charge of dealing with pragma directives. The pre-processor takes in all source and header files to generate what we call pre-processing translation units (.i files). These pre-processing translation units get sent off to the next phase of the compilation pipeline, so we typically never see them.

Compilation:

Confusingly, compilation is one of the steps within the compilation process. The compiler takes in all of the pre-processed translation units from the pre-processor which have now all had comments stripped, macros substituted, and includes expanded. The compiler has its own very long laundry list of phases within its own pipeline. To give a basic idea, this includes lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation. Essentially, the compiler is in charge of taking in our input language (C) and compiling it to our target language (in our case, assembly). The compiler outputs what we call the “translated translation unit” (.s files) a.k.a. assembly language, and passes it to the assembler.

Assembler:

The assembler is much like the compiler, only for assembly language. If you’ve ever worked with assembly, you would know that it needs to get assembled via an assembler, which breaks down the human readable assembly code into machine-readable code called “machine code”. Effectively, machine code is just binary. Technically, the assembler will actually convert the assembly code into object code (.o files), which is the same as machine code, but contains extra metadata in its header.

Linker:

The linker is an important step in the compilation pipeline. There are two things that the linker does: static linking and dynamic linking. Static linking is concerned with the object files generated by the assembler. It will look for the definition of certain functions and variables and match them to their corresponding address locations. For example, if a function was defined within a.o, but is called in b.o, the linker must tell b.o that the function is defined in a.o. In other words, static linking resolves symbols and relocates code amongst the object files that were output by the assembler. I mentioned earlier that Linux has shared object files (.so). Shared object files are typically libraries (a collection of .o files) that are loaded into main memory for any program to access. For instance, almost all programs link the C runtime library. On Unix systems, the C runtime library is known as libc. In GNU/Linux, the C runtime library is technically known as glibc, since GNU rewrote libc from Unix (On Windows, the C runtime library might be called libcmt.lib, etc.). In other words, the C runtime library can have multiple implementations depending on the platform. Anyways, having libc be a shared object means that we can reduce executable sizes since we don’t need to add the contents of libc to our program. On top of that, if libc were to ever need a rewrite/update, it could do this without affecting your program. In my case, libc is located under /usr/lib/libc.so.6. If we run the file command on it (which outputs information about the type of file), we see that it is an ELF 64-bit shared object file and that it is dynamically linked. To summarize, the linker does both static, and dynamic linking, and is responsible for linking any included libraries as well. The linker will output an executable for us to be able to run.

Loader:

It may be debated as to whether the loader is part of the compilation phase. Whether or not it is, I think it’s important that we learn about it, so I’m including it here. The loader is simply responsible for loading our executable code into memory. When you run ./exe the loader must create a new running process. Your process reserves a location in memory where the program stack (as well as certain system resources that must be accessible to the process) will reside. It will then create a jump instruction to your program so that it can begin execution. Processes are a very complex topic which I plan to cover in the future (likely in the Linux document) so look out for that. Now that we have an executable, we can run the program. Here are some interesting commands to try on the command line: strings, file, readelf, ldd, ltrace, strace. The strings command will print out all ASCII strings within the executable file. Most of the file is binary garbage that we can’t read, but strings will find metadata which may be useful. Note that the only purpose of strings is to show you the strings that exist within the file, nothing more. The file command, as we seen, tells you more information about the file you give it. Primarily it just tells you the type of file, but it may give you a varying degree of information depending on the file type. The readelf utility has a few different command line options which you can check out in the man pages. The -a option outputs all metadata. This utility is similar to objdump, but is specifically for ELF executables. It is most useful for showing us the debug symbols ie. the function and variable names. When we get to debugging, you’ll see that the debugger actually strips these symbols from the executable. The ldd command will tell you what libraries your executable depends on, simple as that. The strace utility shows a traceback of all the system calls that were made by your executable. Finally, ltrace will show a backtrace of all of the dynamic library calls made by your executable.

Objects:

Cleanse your brain of the OOP conditioning which hath been instilled in you – C is NOT object-oriented; it is entirely procedural. When I say object, I am referring to anything volatile which includes functions, variables, parameters, etc. Primarily, we will be focusing on variables in this section. We can think about an object in 3 different ways: The storage duration of the object, the scope of the object, and the linkage of the object. I will describe all three of these in detail, and then we can go over some examples.

Storage Duration:

The storage duration has to do with the object’s lifetime i.e. the period of time during which an object has a fixed address and retains the last value that was written to it. There are 4 types of storage duration:

**static:** Exists for the life of the program and its address never changes

**automatic:** Objects with a local scope, such as functions, register contents, or stack allocated variables

**allocated:** Variables which are allocated in heap space using something like malloc(), calloc(), realloc(), mmap(), brk(), srbk(), etc.

**thread:** Exists for the duration of the local thread

We will mostly be ignoring the thread duration, so we just need to remember that static means that the variable lasts for the duration of the program, automatic means that it lasts for the duration of the enclosing scope, and allocated means that it lasts until we call free() (or the process terminates and the OS retrieves it).

Scope:

The scope of an object refers to its “visibility” i.e. what the object is able to access and what other things are able to access the object. There are 4 scopes that we can attribute to an object:

**function:** The word function is quite misleading here. What we really mean in this context are “labels” as in “goto” labels. If you’re not familiar with goto and/or goto labels, I recommend that you do a quick search online, as we really don’t need to cover them in this document.

**block:** A.K.A scope. A region of code between two curly braces { }

**file:** Anything within the enclosing file

**thread:** The thread that has ownership of the object

Linkage:

Linkage refers to where the object is being linked from/to. There are technically 3 kinds of linkage:

**internal:** Only for use within the file

**external:** May be accessed outside the file e.g. libraries or other object files

**none:** Concept of linkage doesn’t make any sense in the context of the object

Storage Classes:

Remember how we just talked about storage duration? Well, C has some keywords known as “storage classes” which give our variables a specific storage duration. There are actually 5 storage classes, though most articles will tell you that there are only 4. Technically, there are only 4, but a fifth was added in C11, “thread\_local”. I will cover all 5 here, but just be aware that thread\_local is not very common. Also note that a variable may only be qualified by 1 storage class (which should be fairly self-explanatory, as they indicate to the compiler the location at which the variable ought to be stored).

**auto:** The auto keyword in C indicates that an object should have an automatic storage duration. In other words, it should live for the duration of the enclosing scope. If a variable has block scope, then it will be destroyed after the instruction pointer moves past the closing curly brace. auto does not need to be declared explicitly in most cases. Any variable declared within block scope will be auto by default. When a variable is declared auto, the compiler reserves the appropriate amount of space based on the data type, but this will very likely be stale data that was left behind. We call this garbage data. Until we actually initialize the variable with some value, the variable will contain a garbage value. For this reason, it is imperative that you initialize automatic variables before they are accessed/read.

**extern:** The extern keyword along with the next keyword confuse many people. In essence, extern is a global value. Whilst the scope of extern is still within the file that the variable was declared in, it is possible to access extern variables from other files. You may declare variables as extern within source files, but it is best practice to declare them within header files. We will go over extern in more detail eventually. extern variables must be *re-declared* in every source file that uses them.

**static:** The other keyword that throws people off is static. The static storage class allows variables to remain in existence for the duration of the program. Its linkage is only local to the file that the variable is declared in. That means that anything that tries to access a static variable outside of the translation unit that it was defined in will fail because the linker was told not allow the static variable to be visible to other translation units. This also allows us to declare/define two functions or variables with the same name technically speaking (although that is not its intended use and will probably break code). static variables are initialized to 0, unlike auto and register keywords, which are uninitialized, and therefore contain garbage. Another way of thinking about static in C is to think of it as meaning private. Interestingly, static functions must be declared inline, meaning that we don’t declare the function in a header file like with normal functions, but instead, declare and define the static function at the same time within the source file. The best way to think about the static qualifier is that it effectively makes your variables and functions “private”. This is useful if you’re creating some sort of API and have functions that need to be accessed internally by other functions in the API, but that shouldn’t be exposed to the end user.

**register:** The register keyword is seldomly used, but we will cover it anyways. register will make a *suggestion* to the compiler to put a variable within one of its registers (we have no control over which register or even if it actually will get placed in a register). This is useful in the case that we know that the variable is about to be used a *lot*. Be humble though, the compiler is way smarter than you, and can optimize better than you, so in all likelihood, it will already have placed the highest priority items in the CPU’s registers. register is barely ever used because the compiler will typically optimize this for you anyways, so don’t worry about it too much. Just know that it exists, and perhaps you may even stumble across it in some production code one day if you’re lucky ;) In terms of limitations of the register keyword, note that variables which are declared within a global scope (i.e., outside main() or any other block) is prohibited, as well as storing the address of a variable via the ‘&’ operator.

**thread\_local:** As of C11, the thread\_local storage class was added. The main idea is that the object will last for the lifetime of the thread. If a thread dies, then so do the variables within it – simple as that! The objects value is initialized when the thread is started and cleaned up when the thread dies.

Storage Duration, Linkage, Scope:

We’re going to go over a bunch of variable/function declarations/definitions and determine each of their storage duration, linkage, and scope:

**// Storage Linkage Scope**

int a; // static duration, external linkage, file scope

static int b; // static duration, internal linkage, file scope

extern int c; // static duration external linkage, file scope

thread\_local d; // thread duration, none, thread scope

void foo(int e) { // automatic duration, none, block scope

int f; // automatic duration, none, block scope

static int g; // static duration, none, block scope

extern int h; // static duration, external linkage, block scope

}

Hopefully most of these are apparent to you. The first 4 of variables are declared outside of any block, meaning that they must all have file scope (other than thread which is unique). The second set of variables *are* declared within a block, so they have block scope. All the static variables have internal linkage, and all of the external variables have external linkage, otherwise the linkage is none. If you are wondering why the first int has external linkage, it is simply because that is the default for variables declared outside of a block, similar to how auto is the default for variables declared within functions. As for storage duration, this mainly depends on whether or not the variable was declared within a block. If it was declared within a block and has no special storage class, then it is auto, otherwise it will be static (with the exception of thread\_local variables).

Type Qualifiers:

I feel that now would be a good time to go over type qualifiers since they sort of tie in with storage classes. Type qualifiers are special keywords that we can put in front of a variable which will tell external sources to treat it in some special manner. There are three type qualifiers, and they don’t really have much in common with each other, other than the fact that they “force” external resources attempting to read or write to abide by certain principles. These three keywords are const, volatile, and restrict.

const:

The const keyword is common in many programming languages. In Java, const is known as “final”. const stands for constant, meaning that the value of the variable should not be altered. While const is a very good safeguard, it is certainly not bullet proof. There are still ways to forcibly write a new value into a variable that is marked const. Often times, beginners get confused as to what the differences are between const variables and macros. The difference between macros and variables marked const is quite significant, and we need to be careful not to conflate the two. Firstly, macros are expanded during the pre-processor phase, while const values are evaluated at compile time. Secondly, macros do not have a data type, but const variables do. Thirdly, macros cannot be altered unless they are #endef(ined) and #defined(d) again. const variables on the other hand, can be forcibly altered using what we call pointers (which we will get into soon enough). When we get to arrays, you will note that their size is always either specified with a macro, or a numeric literal. This is because the compiler can trust that the macro will not change between compile time and runtime, but it cannot make the same guarantee for variables marked const, since a const variable could technically be altered prior to the initialization of the array. While it can be tedious to add const everywhere, it should be used judiciously for variables and parameters that ought not be altered.

volatile:

The volatile keyword is quite interesting – albeit a bit niche. The compiler attempts to optimize variables under the assumption that the only thing which can alter their value are other resources which are handled by the main thread of execution. However, there are external resources (typically other threads) which can alter the state of a variable without the main thread’s knowledge. The volatile keyword tells the compiler not to make such optimizations on the variable. It says: “you *must* check the value of this variable each and every time that you read from it”. The main use for volatile is for multi-threaded programs, as I’ve already implied. If another thread alters the state of a variable, then we want to be reading that variable every time expecting a change. Outside of multi-threading, volatile is not heavily used, so don’t worry about it too much. A good example of volatile in case you were wondering, is when dealing with condvars. If you’re not familiar with what those are, worry not, as we will cover them later in the multi-threading section.

restrict:

We haven’t discussed pointers yet, as I’ve already mentioned, but in short, a pointer is a variable that acts as a pipe/tunnel towards another variable’s data. By accessing the pointer, we can access the contents of the variable that it points to. restrict is given to a pointer and tells the compiler that the pointer is effectively the sole “owner” of the value that it points to. This means that no other pointers shall point to the same data. This is good in most circumstances, as we do not typically want multiple pointers pointing to the same data. The compiler will reduce certain optimization instructions in order to accomodate for the fact that there is a restrict pointer. Note that restrict is more of a promise made to the compiler rather than a strict enforcement. You can still accidentally mark a pointer as restricted but fail to adhere your own suggestion, which can actually then decrease performance, as the compiler will have to correct for your mistakes.

Memory:

This will likely be another gigantic section where I ramble on about memory. Memory is sort of the essence of C, so it is important that we understand how it works. Over the years, there have been many, many optimizations to memory. I don’t plan on covering all of these, but I do plan on giving you a good fundamental understanding of some of the tricks used by modern PCs, because when it comes to pointers and memory manipulation, we need to be very comfortable with this stuff. We will begin with the CPU of all things and work our way to main memory.

Refresher on CPU Architecture:

If it’s been a while since your Operating Systems course (or better yet, my Computer Hardware document), then allow this to be a bit of a refresher, as well as a potential area for learning new concepts. We will skip over microprocessors and just focus our attention on modern day processors. These days, nearly all CPUs have at least two physical cores. These physical cores are identical to each other – that is to say they are designed as twins (physically identical). Each core contains a set of registers (32 registers is typical for modern day CPUs). If your confused about what a register is, it is really just another form of memory located on the CPU. In fact, any hardware that is capable of storing data can technically be classified as memory, which includes hard/solid state drives as well (hence why we can use them as swap memory). Registers are designed to be very small (64 bits or 8 bytes on x86\_64) i.e. they cannot contain a lot of data, but the benefit is that they are extremely close to the Arithmetic Logic Unit (ALU), Control Unit (CU), and Memory Unit (MU). This means lighting fast calculations and read/write times. A bit further away (physically) from the registers, are the CPU’s caches. A cache is also just memory. In modern processors, there are typically 3 levels of cache: L1, L2, and L3. Usually, L3 is shared between 2 cores, but the layout really depends on the architecture of the processor. Caches are also designed to be small, and they actually become larger in terms of storage size as you get further (physically) from the ALU. L1 is the smallest, then L2, then L3. Although our program’s call stack always remains in Main Memory, away from the CPU, data which is accessed will often get copied to the CPU caches to increase speed. Data which is less frequently used will be demoted to lower and lower cache levels in order to leave room in the higher access speed caches for variables that are more frequently accessed. When the CPU searches an entire cache and does not find the data that it is looking for, we call this a “cache miss”. The L1, L2, and L3 caches are further broken down into sectors called cache lines, which are the smallest segment of data that can be mapped into a cache. Note that registers and caches are both local to the CPU, but main memory is not. One other detail to note is that some CPUs have a dedicated icache, which stands for instruction cache. L1, L2, and L3 are usually used exclusively for data (dcache), but the CPU might also have a separate cache for CPU instructions that will also populate with instructions which are executed frequently.

The MMU:

The Memory Management Unit (MMU) serves two primary purposes. The first purpose is to act as a sort of bodyguard that guards data transactions between micro controllers/processors and tries to prevent accidental, or even malicious, memory manipulation. The other purpose is to map physical memory to a virtual address space. You may have heard that there exists both virtual and physical memory. Technically speaking, the correct term for virtual memory is Virtual Address Space (VAS) and the correct term for physical memory is Physical Address Space (PAS). Before MMUs existed, we only had PAS. This led to numerous issues. VAS, on the other hand, is purely conceptual – it does not reside anywhere physically. The MMU makes up an address space, let’s say 0x00000000 to 0xFFFFFFFF, which will be our VAS. The MMU then maps “pages” of VAS to PAS. Whereas PAS is raw (just a contiguous block of memory), VAS is segmented into sectors that are typically 4096 bytes in size (4KB) called pages. Each page has a base address which will be the first address of that respective page. This is useful for a few reasons. The first reason is that it is much easier for us to get contiguous memory in VAS. You see, back in the days where VAS did not exist, memory was “static”. We had to pre-allocate certain locations in memory for our program, variables, and I/O. This led to easy attacks because hackers would be able to know exactly where everything started and ended and could write or read from the correct locations to perform malicious attacks. The primary solution to this is called Address Space Layout Randomization (ASLR). ASLR scrambles our processes’ virtual addresses randomly, so that the pages are all scattered, and writing or reading from somewhere is more likely to result in a crash. The MMU also detects access violations. It does this by noting the virtual address spaces for each process and ensures that processes do not attempt to read/write to/from another processes virtual address space. One last note about the MMU is that it has a lookup table called the Translation Lookaside Buffer (TLB). This buffer, similar to cache in the CPU, stores the most recent address mappings. This is done for efficiency. If the TLB did not exist, each time that we tried to access a virtual address, the MMU would then have to do a linear search to find the corresponding physical address. The TLB stores recent mappings so that it need not go far to find the mapping.

The Stack:

You’ve likely heard of something called the stack, or stack space. Certain embedded systems might not implement stacks at all, and program stacks are not even mentioned within the C specification. However, in most modern systems which are not embedded, it is probably safe to assume that each program executes within its own unique program stack. The MMU is in charge of figuring out approximately how much space the program will require, and then reserving space for it by mapping VAS to PAS. Stacks can be quite large depending on the program. The benefit of stacks is that they isolate a program to a specific address space. If the program attempts to read or write from a location outside of this stack space, the MMU will throw an error and terminate the program. This doubles a security feature. The stack is further segmented into parts called stack frames. Stack frames, for all intents and purposes are equivalent to sub-routines of the program. When a sub-routine call is made, a new stack frame is created, and we push the appropriate data onto that frame. The sorts of data include other function calls, saved registers, local variables, the frame pointer of the function being called, and arguments to the function. If you recall going over storage duration, you’ll hopefully remember that automatic variables are stored for the duration of their enclosing scope, which really comes down to the stack frame they belong to. Once the function ends, the stack frame is popped off the stack, effectively making that memory available once more. Because stack frames are local to the program stack, data can be accessed quickly between frames. The downside to stack frames is that they are volatile, since they are popped off i.e. destroyed.

Heap Space:

Sometimes we require a large chunk of memory. Unfortunately, the stack cannot provide us with much memory because the MMU only reserves so much prior to the program beginning its execution. This is where heap space comes into play. Heap space, also known as ‘the heap’, is a large memory pool located somewhere in RAM. The actual implementation of heap space depends on the system. On older systems, heap space might have been memory shared by all processes. On modern systems, a certain amount of RAM is reserved as heap space on a per-process basis next to stack space. We can request to reserve some of this memory with a few functions that we’ll look at later on. Unfortunately, since heap space is not local to our stack, it is slower to access, and is less ideal than making local allocations. Sometimes we have no option, however.

Pointers:

The worst part about C for most programmers is that they must manage memory. And it’s a fair complaint, because it is certainly not second nature to us to have to consider where and how are variables are being stored. However, this downside to C is also one of its greatest assets. Being able to reserve, reallocate, and free memory on command is not something that many programming languages allow you to do. Having this sort of control can be very useful. Enter pointers! As we’ve already discussed, a pointer is a variable that contains an address to another variable. It is very important that you drill that into your brain early on. A pointer cannot contain a char literal, or an int literal, or a float literal, or any value other than a valid address in hexadecimal. None-the-less, pointers still have data types. The reason that pointers have data types is because pointers can only point to variables (or memory) which have the same data type as the pointer. For example, a char pointer can only point to char variables. It should not point to an int variable. This is a protective measure, but we will see that it is actually possible to have pointers that lack a data type meaning that they can point to any kind of variable.

Before we continue with creating pointers, we must learn two new operators in C. The address operator (&) and the dereference operator (\*). These operators are somewhat confusing to new C programmers because they both serve multiple purposes. The address operator can also be used as a logical AND, and the dereference operator can also be used as the multiplication operator. The way that C differentiates between them is based off the context in which you use them. Both the address operator and dereference operator must be placed in front of a variable e.g. &var, \*ptr. In this case, &var will return the hexadecimal address in VAS of the variable var. The dereference operator only works on pointers. It means “give me the value being held by the variable that you’re pointing to”. This will make more sense when we get to examples. What makes the dereference operator even more confusing is that it takes on a different meaning when defining pointers. The way that we would declare/define a pointer is as follows: <data type> \* <pointer name>; In the context of defining/declaring a pointer, the dereference operator just indicates to C that the following variable will be a pointer. It does not mean “dereference the pointer”. The dereference operator can be placed anywhere between the data type and the pointer name. All three of the following are valid:

int\* ptr; // On the left

int \* ptr; // In the middle

int \*ptr; // On the right

Each of the above means “create an integer pointer called ptr. If anytime after the declaration of ptr we use the dereference operator \*ptr, then it acts as we would expect, and dereferences the pointer ptr.

Examples of Creating Pointers:

Let’s look at how we create a pointer and how we actually point to another variable:

#include <stdio.h>

int main(void) {

int num = 10;

int \*ptr = &num;

printf(“Value of num: %d\n”, num); // Prints 10

printf(“Address of num: %p\n”, &num); // Prints 0x7ffc2eb8bc9c

printf(“Value of ptr: %p\n”, ptr); // Prints 0x7ffc2eb8bc9c

printf(“Address of ptr: %p\n”, &ptr); // Prints 0x7ffc2eb8bca0

printf(“Value that ptr points to: %d\n”, \*ptr); // Prints 10

return 0;

}

In this program I create an int called num and give it the value 10. Then I create an int pointer called ptr. Note that I use the address operator to return the address of num. Since num and ptr are both of type int, ptr is allowed to hold the address of num. Now let’s dissect the print statements. In the first print statement, I pass in num as an argument. This will obviously just print num’s value which is 10. In the second print statement, I give num as an argument once again, but this time I use the address operator in front of it to return num’s address. In my case, its address was 0x7ffc2eb8bc9c. Next, I want to print out the value that ptr contains. Since ptr contains the address of num, this prints out the same address that we had seen previously. Note though, that ptr has its own address in memory, since we need to be able to store it somewhere as well. Therefore, I print the address of ptr using &ptr. This is a different address than that of num, naturally. Finally, we use the dereference operator to dereference ptr. That will return the value associated with the address that ptr contains. ptr contains the address of num, so we return 10 since the value 10 is stored at the address 0x7ffc2eb8bc9c.

More on Pointers:

Pointers are an essential concept in C, so I would like to continue exploring them. First off, let’s discuss the size of a pointer. On modern 64-bit machines, a pointer will always be 8 bytes (on a 32-bit machine, a pointer is 4 bytes). This does not mean that the value being pointed to is 8 bytes (which is important to understand). For instance, char pointers are used to point to string literals. I may have a string literal or even a long double, which occupy more than 8 bytes. That is okay, because the pointer only points to the base address of the data, it doesn’t contain the data itself (remember – pointers ONLY store ADDRESSES)!

Pointer Arithmetic:

In C, we can perform arithmetic on pointers. But what does it mean to add/subtract/divide/multiply an address? C has defined a special behaviour for performing arithmetic on pointers. If for example, you were to add 1 to a pointer e.g. ptr++; it would not, in fact, add the number 1 to the address. Instead, it would add 1 \* sizeof(data type). So, if ptr was an int, and int was defined as 4 bytes, ptr++ would add 0x4 (hexadecimal 4). This effectively means that the pointer no longer points to the original int, because the address that ptr contains is now *offset* by 0x04. This is useful for arrays as we will see, since we can continuously get the next/previous item in the array by doing ptr++ or ptr--; Using multiplication and division are rarely used on pointers because you will likely end up with an address that is way outside the boundaries of the stack.

Double Pointers:

In case the thought hasn’t crossed your mind yet – yes, we can create pointers to other pointers. This also tends to confuse people, but it’s really not that bad after a bit of practice. If we define a pointer as char \*ptr1, then we can create what’s called a double pointer to point to ptr. This would be defined like so: char \*\*ptr2 = &ptr1; Remember that ptr has its own address, so we can simply pass it to the double pointer using the address operator. Note the double dereference operator to denote that it’s a double pointer. The more asterisks, the more “layers” you add. When I’m explaining this to new C programmers, I like to tell them that this is akin to an onion, and every time you dereference the pointer, you peel off a layer, inching closer towards the value located at the core. In order to dereference ptr2 entirely, we would use the dereference operator twice like so: char c = \*\*ptr2; The first dereference returns ptr1, and the second dereference returns whatever ptr1 was pointing to. In all likelihood, the highest level of pointer that you’ll ever see is a triple pointer, but even those are quite uncommon.

Void Pointer:

Let’s talk about void pointers. Remember how I said that we could have pointers that point to any data type? This is achieved by using void pointers. A void pointer is created as follows: void \*void\_ptr = &data; void pointers are both useful and dangerous. They allow for versatility when passing in data since we can pass data of any data type to them, however, it is possible that we pass in one kind of data and interpret it later as a different type of data, which is usually not the desired effect. Void pointers should be treated as raw bytes that can be cast back to anything. For example, take the following code:

#include <stdio.h>

/\* Function declaration (Don't need argument names) \*/

double cast\_to\_double(void \*);

/\* Function definition \*/

double cast\_to\_double(void \*arg) {

double \*dp = (double \*)arg;

return ( \*dp );

}

int main(void) {

char c = '\0';

char \*cp = &c;

double mystery = cast\_to\_double(cp);

printf("Mystery value is %f\n", mystery);

return 0;

}

We shall run through this together. Program execution begins in main. We create a char called c which contains the special character ‘\0’. Technically ‘\0’ is not the same as the number 0, since ‘\0’ is escaped with ‘\’. I create a pointer to c called cp. Then we create a double called mystery and set it equal to the return value of the function cast\_to\_double, passing in cp as an argument. As an aside, you may have noticed that I added a comment on the declaration of the function cast\_to\_double() where I pointed out that in function declarations you don’t have to specify the parameter names if you don’t want, only the data types of each ordered parameter. In the actual definition you *do* need names however, and in most cases I recommend that you put the names in for both the declaration and definition (the only reason I didn’t in this case is because the declaration isn’t in its own header file as it normally would be). Anyways, the parameter arg is a void pointer, which means that we can pass in an address that points to any kind of data. We pass in the address being held by cp, which happens to be the address of c. An implicit cast occurs here where cp becomes a void pointer and gets stored in arg e.g. void \*arg = (void\*)cp; Within the function we create yet another pointer called dp which is set to the address held by arg (still &c). This is an explicit cast; whatever data associated with the address passed to dp is now treated as if it were a double. We return the value held by dp by dereferencing it and print it to the console (stdout). As it turns out, this program prints “Mystery value is 0.000000”. But what was the ‘\0’ that we actually passed to the function anyways? This brings us to the next topic.

NULL:

As a new C programmer, it is very important to understand what NULL is, considering that it is used frequently throughout our code. There are many debates as to whether or not NULL is a good thing within the programming space in general, but all of those debates are moot when it comes to C, because NULL is a C staple; one which will never be replaced in the language. First, let’s discuss which header files define NULL. According to the C standard, NULL must be defined in locale.h, stddef.h, stdio.h, stdlib.h, string.h, time.h, and wchar.h. Essentially, inclusion of any of these header files gives us access to NULL. Note that NULL is capitalized, indicating that NULL is defined as a macro in C, which happens to be the case. The actual implementation of NULL can vary depending upon which version of the C standard library you’re using. In glibc, NULL is defined as (void \*)0. In other words, it is a pointer to address 0x00000000 in memory. Of course, if you’re an experienced programmer, you may know that that is a virtual address, which does not necessarily correspond to the actual physical address 0x00000000. Just as (void \*)0 is a valid literal that denotes NULL, we too can use the special character ‘\0’ or just the number 0 to represent NULL as well. But if strings are NULL terminated, and the number 0 is equivallent to NULL, how then are we able to have zeros within our strings, you may be asking yourself. If you had that thought, it’s because you failed to recall that the character literal ‘0’ is not actually the same as the number 0. The character ‘0’ in ASCII is decimal 48, whereas the character ‘\0’ is actually the number 0.

It is considered a best practice to avoid declaring pointers without initializing them. This is because if we create a pointer (which will presumably have automatic storage duration) and don’t give it an initial value, then it will point to garbage which it will interpret as a hexadecimal address. To address this, we like to set pointers = NULL if we are not ready to point them to something yet. This has the additional benefit of allowing us to do error checking. One other thing to note is that if we set a pointer equal to the address of a variable that is within a block scope, and that scope exits (thus destroying the variable that was being referenced) then we now have what’s called a null pointer. This is perhaps a confusing name for it, since a null pointer is not actually a pointer that points to the NULL macro, but rather one that points to invalid memory (an address which does not belong to our process’ address space). Dereferencing a null pointer leads to undefined behaviour. In most cases you will receive a segmentation fault which will crash the execution environment (program), however, it could be the case that you’re on an embedded system with no MMU, in which case you might begin executing some random code elsewhere in memory (albeit very unlikely it wouldn’t crash). I will show you an example of accidentally creating a null pointer, and how we can error check it.

#include <stdio.h>

/\* Function declaration \*/

void test\_null\_ptr(void \*);

/\* Function definition \*/

void create\_null\_ptr(void \*arg) {

char c = 'a';

arg = &c;

} /\* Local variable c goes out of block scope \*/

int main(void) {

/\* Good practice to initialize ptr with NULL

if we are not initializing it right away \*/

char \*nullptr;

create\_null\_ptr(nullptr);

printf("Value pointed to by nullptr is: %c\n", \*nullptr);

return 0;

}

The above example has no error checking, and therefore the program may crash when we try to dereference nullptr in the printf statement. In C, 0 means false, and anything that is not 0 means true (including negative numbers). Therefore, we could tell our program to only dereference nullptr **if** nullptr != NULL. We can simplify this expression to just if (nullptr) meaning, if nullptr is anything but NULL, we are allowed to continue. Note that this will only work if we set nullptr = NULL when we declare it, which we are not doing in the above example. Here is how we could do error handling for the above example then:

#include <stdio.h>

#include <stdlib.h>

/\* Function declaration \*/

void test\_null\_ptr(void \*);

/\* Function definition \*/

void create\_null\_ptr(void \*arg) {

char c = 'a';

arg = &c;

} /\* Local variable c goes out of block scope \*/

int main(void) {

/\* Good practice to initialize ptr with NULL

if we are not setting it right away \*/

char \*nullptr = NULL;

create\_null\_ptr(nullptr);

if (nullptr) {

printf("Address contained in nullptr is: %p\n", nullptr);

} else {

printf("nullptr does not point to an address\n");

exit(EXIT\_FAILURE);

}

printf("Value pointed to by nullptr is: %c\n", \*nullptr);

return 0;

}

Since we set nullptr = NULL, our if condition fails and we go to the second “else” branch. We can also perform the same error checking using assert() (we will review assert() again in a bit). The assert() function takes in some conditional statement and if it evaluates to false the program terminates with an predefined error statement to stderr. Note that we must #include <assert.h> to use this function. This would look like the following:

assert(nullptr);

printf("Value of nullptr is: %c\n", \*nullptr);

/\* Prints the following: \*/

test: test.c:21: main: Assertion `nullptr' failed.

Aborted

Using const Keyword With Pointers:

We’ve already covered the const keyword for regular variables, however, there is another use-case for const when it comes to pointers. Not only can we apply the const keyword to the pointer variable, but also to the value being pointed to by the pointer. I will show you an example, followed by the output of the compiler, and then we will review:

int main(void) {

long lng = 999999999;

long \* const pLng1 = &lng;

long const \* pLng2 = &lng;

long const \* const pLng3 = &lng;

pLng1++; /\* Incrementing pointer pLng1 \*/

(\*pLng1)++; /\* Incrementing variable lng \*/

pLng2++;

(\*pLng2)++;

pLng3++;

(\*pLng3)++;

return 0;

}

/\* Output of the compiler \*/

test.c:10:12: error: increment of read-only variable ‘pLng1’

10 | pLng1++;

| ^~

test.c:14:15: error: increment of read-only location ‘\*pLng2’

14 | (\*pLng2)++;

| ^~

test.c:16:12: error: increment of read-only variable ‘pLng3’

16 | pLng3++;

| ^~

test.c:17:15: error: increment of read-only location ‘\*(const long int \*)pLng3’

In this example, I create 3 pointers pLng1, pLng2, and pLng3 which all point to lng (note that under normal circumstances, you should not have multiple pointers to the same variable). You’ll notice that depending on where we place the keyword const determines what is and isn’t considered constant. pLng1 is defined as long \* const pLng; In this case, const applies to the pointer itself (pLng1) so the compiler throws an error when we try performing pointer arithmetic on it, but it does not complain when we increment lng once dereferenced. pLng2 is defined as long const \* pLng2; In this case, the opposite occurs. The compiler is okay with pLng2 being mutated (in this case, incremented), but not with us trying to increment lng, since it is what is declared const in this case. In the third case with pLng3, the compiler is not okay with the pointer being incremented, nor with lng being incremented via the pointer.

This syntax can appear quite confusing for new C programmers, but it is actually not too bad once you understand that what matters is which side of the kleane star (\*) that the const keyword lies. If const lies to the left of it, it applies to the pointer. Likewise, if const lies to the right of the kleane star, it applies to the memory which is accessed when dereferenced. With this in mind, we can recognize that the following two statements are identical:

const void\* ptr;

void const \*ptr;

Function Pointers:

That is correct, pointers *do* in fact get scarier! But frankly, function pointers are just regular pointers that point to functions. This is possible because function names are literally just labels that represent addresses (exactly the same as variable names) - addresses to regions of executable code. The typical use-case for functions is to create a function table i.e. an array of functions, and then have one function pointer which can point to one function in the function table at a time, and change what function it’s pointing to based on some branching logic. Another common use of function pointers is when creating threads, as each thread can take in a function pointer as a parameter so that that thread can operate on that specific function (more on that in the multi-threading section). The syntax for function pointers is a bit off-putting, but it’s not that bad after a bit of getting used to. The data type of the pointer should match the return type of the function. Therefore, if the function returns void, the pointer will be of type void as well. The pointer must also take in a list of comma-separated data types which are meant to align with the data types of the parameters in the function being pointed to. For example, a function with the following signature: long foo(int a, char b); could be pointed to with a function pointer akin to: long (\*pFoo)(int, char) = &foo; Is this sort of confusing? Yes... But it also makes sense if you squint at it for a while. There are two ways to call a function via a function pointer. The first, and more complicated way, is to dereference the function pointer like so: (\*pFoo)(0, ‘w’); This makes it clear to the programmer that this is a function pointer call, however, I find this to be a bit confusing, personally. The easier way is to simply call it as if it were a regular function, e.g. pFoo(5, ‘g’); Ex.

int foo(int, char);

int foo(int a, char b) {

return (a + b);

}

int main (int argc, char \*argv[]) {

int (\*pFoo) (int, char) = &foo;

/\* First method of calling \*/

printf("Sum of 0 and w is %d\n", (\*pFoo) (0, 'w'));

/\* Second method of calling \*/

printf("Sum of 5 and g is %d\n", pFoo(5, 'g'));

return 0;

}

Although we haven’t really gone over arrays yet, the following should make enough sense to you, assuming that you have some knowledge of how arrays work. It is crucial to make note that function pointer arrays/tables only work if all the functions in the table share the same signature. Assuming that we have a few functions that all share the same signature, we create the function table by creating a normal function pointer. The only difference this time is that by adding the square brackets, we are indicating that we are creating an array of function pointers. We can then initialize this array with something called an initializer list, which will be covered in the array section. Here is an example taken from Geeks for Geeks:

#include <stdio.h>

/\* Function prototypes \*/

void add(int, int);

void subtract(int, int);

void multiply(int, int);

void add(int a, int b) {

printf("Addition is %d\n", a+b);

}

void subtract(int a, int b) {

printf("Subtraction is %d\n", a-b);

}

void multiply(int a, int b) {

printf("Multiplication is %d\n", a\*b);

}

int main(void)

{

/\* fun\_ptr\_arr is an array of function pointers \*/

void (\*fun\_ptr\_arr[]) (int, int) = { add, subtract, multiply };

unsigned int a, b, c;

a = 15;

b = 10;

printf("Enter Choice: add [0], subtract [1], multiply [2]\n");

c = (int)(getchar() - ‘0’); /\* Subtract character ‘0’ (48 in decimal) to get numeric value of input \*/

fflush(stdin); /\* Flush the newline char left by getchar() \*/

if (c > 2) {

printf(“Invalid input. Exiting...”);

exit(EXIT\_FAILURE);

}

(\*fun\_ptr\_arr[c])(a, b);

return 0;

}

Function pointers get much, much more confusing :( Let’s say that we want to pass a function pointer as the argument for another function. Here is an example of how we can do that:

int foo(int (\*bar)(int, int));

In the example above, we declare a function called foo(). It contains a named parameter, bar. The named parameter bar is a function pointer that returns an int and accepts 2 ints as its parameters. Assuming that made sense, I’ll confuse you even more. Instead of accepting a function pointer as an argument to another function, let’s say that you want to return a function pointer from another function. I’ll just show you the syntax and try my best to explain:

int (\*foo(void))(int, int);

Okay, what’s happening here? Well foo is the name of the function that will return the function pointer. Unlike in the previous example where we have a named parameter, we don’t provide a name for the function pointer that we are returning, hence why you don’t see bar referenced anywhere. The return type is indicated by everything surrounding the declaration of foo(). So the initial int indicates the return type of the function pointer, and the two ints enclosed in brackets at the end are the argument list for the function pointer. It is often advised that you use a typedef in either of these cases. Here is an example (see if you can follow along):

#include <stdio.h>

typedef int (\*add\_t)(int, int); /\* Create a typedef called add\_t \*/

int add(int a, int b) { /\* A normal function which returns the sum of a and b \*/

return a + b;

}

add\_t retFuncPtr(void) { /\* This returns the address of add() as an add\_t type \*/

return &add;

}

int acceptFuncPtr(add\_t fptr, int a, int b) { /\* This accepts an add\_t and two numbers \*/

return (\*fptr)(a, b); /\* Return the result of the function pointed to by fptr \*/

}

int main(void) {

int a = 2;

int b = 3;

int result = acceptFuncPtr(retFuncPtr(), a, b);

printf("Result of %d + %d = %d\n", a, b, result);

}

Output: Result of 2 + 3 = 5

In case you were curious, here is the same code without the typedef:

#include <stdio.h>

int add(int a, int b) {

return a + b;

}

int (\*retFuncPtr(void))(int, int) {

return &add;

}

int acceptFuncPtr(int (\*fptr)(int, int), int a, int b) {

return (\*fptr)(a, b);

}

int main(void) {

int a = 2;

int b = 3;

int result = acceptFuncPtr(retFuncPtr(), a, b);

printf("Result of %d + %d = %d\n", a, b, result);

}

Because I’m feeling super sadistic and want to make you C noobs cry, I’ll show you a double function pointer just for fun ;) It’s actually not too bad, we just add an additional star to the function pointer like so:

int add(int a, int b) {

return a + b;

}

int acceptDblFuncPtr(int (\*\*fptr)(int, int), int a, int b) {

return (\*\*fptr)(a, b);

}

int main(void) {

int (\*add\_ptr)(int, int) = &add;

int result = acceptDblFuncPtr(&add\_ptr, 2, 3);

printf(“Result of %d + %d = %d\n”, a, b, result);

}

Recall that with a normal function pointer, we can either invoke it by just calling the name e.g. fptr(2, 3); or by explicitly dereferencing it e.g. (\*fptr)(2, 3);. With double function pointers, the former syntax (invoking it by name alone) no longer works. However, we can now optionally dereference it once or twice and it will work the same. In other words, we can invoke a double function pointer like this: (\*fptr)(2, 3); or like this (\*\*fptr)(2, 3);, but not like fptr(2, 3);. This rule applies for triple function pointers and beyond. What can I say, C is weird sometimes!

Operator Precedence:

I’m sure that you’re already aware of operator precedence, but I wanted to talk specifically about the dereference operator. In the example above, we did something like (\*lp1)++; These brackets were necessary because postfix operators have precedence over unary operators. In other words, if we had just done \*lp1++; it would have been parsed as \*(lp1++); which would add 0x8 to the address contained in lp1 and then dereference the contents of that address. In the two unary operators, the compiler reads them left to right. For example, \*++lp1 is read as \*(++lp1) and ++\*lp1 is read as ++(\*lp1).

Returning Persistent Data Using Pointers:

Something that really tripped me up a lot as a beginner and even intermediate C programmer was the marriage between pointers and functions. How ought you return persistent data from a function? We know that we require pointers since local variables are only local to their enclosing scope, but aren’t pointers also local to their enclosing scope as well? Yes! (they are). This is why we have two and only two methods of returning persistent data from a function. Remembering these will spare you much confusion.

1. Don’t return anything

2. Return a reference to *heap*-*allocated* memory

Let’s look at each of these. The first method – “Don’t return anything”? How does this make sense? Well you are already aware that pointers can be passed into functions as parameters. Modifying a local parameter which is a pointer will write to the same shared location in memory that is being referenced by the variable outside of the function (the argument that we “passed in” i.e. created a duplicate reference to). This means that the data will persist. The benefit of this is that it allows us to return something other than the pointer e.g. an error value, and therefore, we can kill 2 birds with 1 stone.

Method 2 is to return *heap-allocated* memory. I emphasize this because we can absolutely never return a pointer to a stack-allocated variable that belongs in the same scope as the pointer. It may seem as though the memory could not persist using this method since the pointer will get destroyed once the stack pointer exits the function’s scope, however, the return value is what is key here. Since we return the address to the location of the memory in the program’s heap (which is persistent for the duration of the program), we can create a new reference/pointer to point to the returned value.

Here is an example of each method:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int \*foo();

void bar(void \*data);

int \*foo()

{

int \*data = malloc(sizeof(int));

\*data = 12345;

return data;

}

void bar(void \*data)

{

strcat(data, "bar");

}

int main(void)

{

char str[100] = "Modified by: ";

int \*num\_p = NULL;

bar((void \*)str);

num\_p = foo();

printf("str = %s\n", str);

printf("num\_p = %d\n", \*num\_p);

free(num\_p); /\* Note that num\_p must be freed because it was heap allocated \*/

return 0;

}

Other Pointer Tips:

Here are some general tips when using pointers to keep in mind:

1. Try to reduce the number of pointers that reference the same memory location. This is not a dogmatic recommendation, but reducing the number of things that can modify the same data leads to code cleanliness and reduces the risk of data redundancy (useless duplicates).

2. Passing pointers is preferable for large data. This primarily applies to structs and arrays. Even if we do not require the changes to a struct or array to be persistent, it still may be beneficial to pass the data by reference rather than by value. This is because the size of a pointer is always 8 bytes, whereas a struct or array might be much larger. Copy operations are expensive, and so passing a struct that is 64 bytes in size to a function will be slower than passing the address of the struct.

Comments:

I can’t believe we haven’t even gotten to comments yet! I’m not sure where to put this section, so I’ll just put it here and keep it brief. All that you need to know about comments is that C++ style comments i.e. // was not introduced until C90. In order to keep code as portable as possible, it only makes sense to use the standard comments for C which are /\* \*/. I have been using C++ style comments in some of the code that I’ve shown thus far because I’m assuming that you started with Java and are more comfortable with them, but C-style comments should be your preferred choice in my opinion.

Type Casting:

It is important that you understand typecasting, as it comes up very frequently in many codebases. As I alluded to in the section on implicit vs. explicit, type casting is an explicit way of translating a variable’s data type to another type. The C compiler does this implicitly for us if we pass in an int to a function that expects a long for example. If the compiler is configured to give us warnings, it will often notify us whenever we have implicit type casts because they often come across as unintentional. Explicit type casts help with readability because they show the programmer that yes, you did, in fact, mean to turn that long into an int. In order to typecast, we simply put the type that we want to cast *to* in parentheses e.g. (int), followed by the variable that we’re type-casting e.g. (int) a\_long; This doesn’t actually alter the type of a\_long. Under the hood, the C compiler is allocating space on the stack for a new int variable and then assigning the contents of a\_long to it. This would like like: int tmp = a\_long;   
  
 Now what if we cast a type with 16 bits such as a short to a type with less bits such as a char. In C, this will truncate the higher significant bits. So, for example, if our short is set to AA BB in hexadecimal, then casting it to a char will mean that it gets truncated to 00 BB. Now technically, I just lied, because if you typecast short to char like this: (char)a\_short; then you will actually end up with FF BB. This is because char is signed by default. We can fix this by either typecasting to unsigned char e.g. (unsigned char)a\_short; or by using something like uint8\_t. Here is a small code snippet to demonstrate:

#include <stdio.h>

#include <stdint.h>

int main(void)

{

short shrt = 0xAABB;

printf("0x%hX\n", (uint8\_t)shrt);

return 0;

}  
  
The code here is pretty self explanatory, except for maybe the weird printf statement. All that’s happening there is that we print “0x” followed by the format specifier h (which means short) and then X (which means print this in hexadecimal notation). The output is 0xBB (which is really 0x00BB).

Casting a pointer to a pointer of another type will effectively alter how the data being pointed to is interpreted. For this reason, C is not considered typesafe. It is undefined behavior within the C specification to cast a data type to a pointer of a different data type with the exception of void \*. For example, I may want to cast a variable or literal into an int pointer, or a char pointer. We can accomplish this like so: (char \*)a\_char; This effectively does the same thing as a regular cast, but creates a temporary char pointer to store the address of a\_char. The compiler would do something like: char \*tmp = &a\_char; You must be cautious because the temporary pointer’s lifetime is only that of the current line being executed. To preserve this temporary variable, either assign it to another char pointer, or pass it as an argument to a function. For example: char \*cp = (char \*)a\_char; The compiler effectively does this: char \*tmp = &a\_char; followed by this: char \*cp = tmp;   
  
 One final trick that I wanted to show, just because I thought it was cool the first time I seen it, is that we can cast initializer lists into arrays (we will cover initializer lists shortly). This is useful when passing values to a function that accepts an array as one of its parameters. For example, given a function prototype like the following: void foo(const int bar[]); we can pass in data like so: foo( (const int[]) ({1, 2, 3}) ); Here, we the compiler is essentially doing const int tmp[3] = {1, 2, 3}; You may appreciate this more as you learn of the struggle that are arrays in C.

Derived Types:

Derived types are low level structures or concepts that are built into C but rely on basic data types to be implemented. These primarily include arrays, strings, structures/structs, and unions. We will look at each of these derived types and how they function.

Arrays:

Arrays should hopefully be a familiar concept to you. They have a fixed size and create key-pair mappings between the indices of the array and the data that we are storing a.k.a. the elements of the array. You must specify the data type of the array since all of the data stored within any given array must be of the same data type. We also throw around the word “contiguous” a lot in C when discussing arrays. It is essential that each index of the array is located next to its neighboring index in memory, thus we say that an array reserves a contiguous chunk/block of memory. You may have been taught in school to think about arrays as a rectangle made up of individual blocks that each represent the indexes of the array. In reality, arrays are not segmented as if we have divisors between each element but are instead just a long sequence of bytes separated by the size of the data type of the array (well in even more detail, the data may be segmented in physical memory, but be mapped contiguously in virtual memory by the MMU). For example, an array such as the following: int arr[5] = {1, 2, 3, 4, 5}; might look like this in memory:

00000000 00000000 00000000 00000001 // 4 bytes reserved for 1

00000000 00000000 00000000 00000010 // 4 bytes reserved for 2

00000000 00000000 00000000 00000011 // 4 bytes reserved for 3

00000000 00000000 00000000 00000100 // 4 bytes reserved for 4

00000000 00000000 00000000 00000101 // 4 bytes reserved for 5

The array (in this case called arr) is essentially used as an alias for the *base address* of the array. The base address of the array is actually just the address of its first element, since that’s where our region of contiguous memory begins. If we want to access some index of the array, it is as simple as adding an offset to that base address. That offset is calculated by taking the size of the data type of the array i.e. sizeof(int) in this case, and multiplying that with the index of the element that we want to access. Therefore, our formula for accessing a particular element in an array is B + si where B is the base address, s is the size of the data type of the array, and i is the index that we want to access. Let us say that we want to access the third element of the array in our example (which would be index 2). The base address of the array might be 0x7ffc3d81ddc0. In order to access the third element, we do 0x7ffc3d81ddc0 + sizeof(int) \* 2

= 0x7ffc3d81ddc0 + 0x4 \* 2

= 0x7ffc3d81ddc0 + 0x8

= 0x7ffe40a5bef0

Now perhaps that was all very confusing to you, but my point is mainly that when we use the index operator (for example, arr[2]), what C is doing behind the scenes is taking the base address of arr and adding sizeof(int) \* 2 to it. Then it simply returns the data located at that address. So, there are no fancy separators between the data of an array. We just have one long contiguous chunk of memory that has a base address, and that we can calculate where elements are located by adding offsets to that base address according to the size of each element within the array.

Creating Arrays:

In order to create an array in C we have two options. The first is the classic way: <data\_type> <var\_name>[<size>]; and the second way is to use an initializer list. Initializer lists can only be used at the time of declaration, and at no point afterwards. Initializer lists are useful when you know what elements will be in the array at the time of creation. For example,

const char \*months[NMONTHS] = { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };

The initializer list works here because we already know what months are going to be within the array.

Let’s quickly talk about the size of arrays. I believe I’ve already mentioned this, but it can’t hurt to say it again: The size of an array must be known at compile time. This means that we cannot have variables (even const variables) to define how large our arrays will be. In other words, macros or explicit integer literals are your only choices here. Note that it is possible to create variable length arrays in C as of C99, but these are very dangerous and should not be used. Another note when using initializer lists is that we do not need to define the size in the square brackets. The size is implicit based off how many elements are in the list. You can, and probably should, still explicitly give it a size though. This helps with readability, and if can help catch errors under some circumstances. For instance, if you provide the array with a size that is less than the number of elements in the initializer list, the compiler will warn you that there are excess elements in the array that will be thrown out.

Examples of Using Arrays:

I feel that the best way to show you how arrays work is to give some code to help you visualize it. The first example I have is a simple program that performs a right bit shift on each element in the array:

#include <stdio.h>

#include <stdint.h>

int main(void) {

int i;

uint8\_t arr[8] = { 128, 64, 32, 16, 8, 4, 2, 1 };

size\_t arr\_size = sizeof(arr) / sizeof(arr[0]);

for (i = 0; i < arr\_size; i++)

printf("%u bitshifted right = %u\n", arr[i], (arr[i]) >> 1);

return 0;

}

// Output

128 bitshifted right = 64

64 bitshifted right = 32

32 bitshifted right = 16

16 bitshifted right = 8

8 bitshifted right = 4

4 bitshifted right = 2

2 bitshifted right = 1

1 bitshifted right = 0

Since I won’t be exceeding numbers past 256 I declare the array to be of type uint8\_t. In this case I explicitly state that the size of the array will be 8 (I could have used a macro here, but I wanted to demonstrate a neat trick in the next line), and I populate it with descending powers of 2. The next line may be somewhat confusing to you, but it helps explain a good point. Since this variable will be representing a size, I give it the typedef size\_t. Then we do sizeof(arr) / sizeof(arr[0]). Hopefully you understand that sizeof() will return the size in bytes of the variable/data type that we give to it. But wouldn’t sizeof(arr) just return the size of uint8\_t since that’s the data type of arr? Well actually, sizeof(arr) returns the size of the entire stack allocation for the array in bytes (since an array is a derived type). This is a unique property of arrays, structs, unions, and in certain cases, strings. sizeof(arr[0]) returns 1 since arr[0] will actually retrieve the initial element of the array and determine that it is of type uint8\_t, which occupies 1 byte. This operation leaves us with the number of elements in the array. If you wanted to visualize that a bit more mathematically:

sizeof(arr) = 8 \* sizeof(uint8\_t)

= sizeof(arr) / sizeof(arr[0])

= 8 \* sizeof(uint8\_t) / sizeof(arr[0])

= 8

We end this function by printing out each element of the array and the result of performing a right bit shift on that element.

Pointers to Arrays:

The relationship between pointers and arrays is blurry, since they share a lot of commonalities. They do differ, but we won’t get into the nuanced differences here. The following is an over-engineered code snippet showing how we can create a pointer to an array:

#include <stdio.h>

#include <stdlib.h>

#define POW\_SIZE 5

int main(void)

{

const int i = 3;

const int pow10[POW\_SIZE] = {1, 10, 100, 1000, 10000};

int const \* const restrict ppow10 = pow10; /\* Note that we don't use the & operator... \*/

printf("Data at index %d = %d\n", \*(ppow10 + i));

return EXIT\_SUCCESS;

}

In this code snippet, we #define the array size for an array called pow10 which contains the first 5 powers of ten. This is unnecessary to do since we use another initializer list, but again, it’s a good safety practice in my opinion. We create a const int called i which will be the index that we access using the pointer. I’ve set it to access the third index i.e. the 4th element. We create the array, pow10, as const since I don’t plan on modifying any of its elements. Finally, we create a pointer to pow10 called ppow10. Both the address that ppow10 contains, as well as the data being pointed to, should remain unmodified, hence the double const. I used the restrict keyword since ppow10 should be the only way of accessing pow10 indirectly. The big take away from this example is that we don’t need to use the address operator when we assign pow10 to ppow10. Unlike a regular variable, C implicitly returns the base address of pow10 when we try to access it without any sort of index operator (square brackets). We *could* use the address operator here if we wanted, and we could even use &pow10[0] if we wanted, but this is often how you’ll see arrays assigned to pointers. Of course, this code prints out “Data at index 3 = 1000”, as we’d expect.

2D Arrays:

2D arrays in C are sort of frowned upon, but not necessarily because they aren’t good practice to use; more so because C doesn’t implement them very well... When we think of a 2-dimensional array, we often envision a matrix/grid which represent rows and columns. This may be a useful way to think about 2D arrays when traversing them with for loops, however, it is better if we can start thinking about 2D arrays from the perspective of memory. In the previous section, we seen that arrays are really just a large section of memory. We call this memory “contiguous” because the addresses in that memory are sequential, as opposed to fragmented. A two-dimensional array is really the same as a one-dimensional array, except that when we get the next “row” index, we are offsetting a multiple of the nested array’s size. When indexing by column, it functions the same as a 1-dimensional array and offsets by a multiple of the data type’s size. Once again, there are no boundaries or boxes, but rather one chunk of contiguous memory with a base address and indeces which are our byte offsets into that memory.

Creating 2D Arrays:

Let’s start simple. A 2D array can be created like so: short arr2D[10][10]; The first size parameter specifies the amount of arrays that the 2D array will be able to contain. The second size parameter specifies how many elements each nested array will be able to contain. Same as normal arrays, we can use initializer lists if we know what elements we want to be in the array at the time of initialization. In order to do this, we do something like the following:

uint8\_t n\_vec2D[3][3] = {{1, 0, 1}, {0, 0, 1}, {1, 1, 1}}; /\* Note the braces enclosed within the outer braces\*/

When using initializer lists in 2D arrays, the first size argument is not required, however, the second size argument is always required. The previous example could be rewritten as:

uint8\_t n\_vec2D[][3] = {{1, 0, 1}, {0, 0, 1}, {1, 1, 1}};

Another thing to note is that if we only provide a single initializer list to a 2D array then all remaining elements get initialized to 0. Careful using this though, because this only works if the row size is specified, otherwise C would not know how many rows to provide. In our previous example, the compiler knew how many rows to allocate because it could count the number of nested initializer lists, however the following example has no such hints.

int repeat[5][5] = {1, 2, 3, 4, 5, 6};

// Outputs:

1 2 3 4 5

6 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

Examples of using 2D Arrays:

#include <stdio.h>

#include <stdint.h>

#define DIM 5

void print\_row(const unsigned row\_num, int arr[][DIM]);

void print\_row(const unsigned row\_num, int arr[][DIM])

{

int i;

for(i = 0; i < DIM; i++)

printf("%d ", arr[row\_num - 1][i]);

printf("\n");

}

int main(void)

{

int int\_arr[][DIM] = {

{1, 0, 0, 0, 0},

{1, 1, 0, 0, 0},

{1, 2, 1, 0, 0},

{1, 3, 3, 1, 0},

{1, 4, 6, 4, 1},

};

print\_row(3, int\_arr);

return 0;

}

Here’s a pretty basic example of just setting up a 2D array using our initializer list and then passing that array in as a parameter to a function. We give the function a parameter of 3 as well to print the third row. This is just a simple for loop that goes through and prints out the values at the appropriate indices.

Pointers to 2D Arrays:

In my opinion, the most confusing that pointers get, is when they point to 2D arrays. In general, you want to avoid doing this, and instead, favor using some sort of pointer that points to a contiguous block of allocated memory while keeping track of the row and column offsets using pointer arithmetic, but I’ll be covering this anyways since it very well might come up in someone’s code. Assume that we have a 2D array called arr2d[10][5] which is of type short, and that it’s been populated with data already. In order to create a pointer to this array, we would do something like: short (\*p\_arr2d)[5] = arr2d; As we look at this, try to remember that C only cares about the “inner” size of the array. We declare the pointer as the same data type as the 2D array (obviously), but this time, we surround the asterisk with brackets. This is to avoid operator precedence issues. If the brackets weren’t there, the compiler would think that we were creating an array of size 5, with each element being a short pointer. Because we have the brackets, the index operator takes on a new meaning. It now treats it like as a pointer to a short array of length 5. Do you see the difference? The first is an array containing 5 pointers, the second is a pointer to an array of 5 shorts. By assigning p\_arr2d with the base address of arr2d, p\_arr2d now essentially points to arr2d[0][0]. Very strange is when we get to pointer arithmetic. Adding 2 for example, will essentially add 2 times the size of the entire 2d array. In this case, our array would have 10 \* 5 = 50 elements so adding 2 would make us jump 100 elements ahead, which is well outside the boundaries of the array. Instead, we first use the index operator to get the nested array that we want, and then add an offset to that (and then dereference). For example, \*(p\_arr2d[2] + 3) would get us the element at row 3 and column 4. If this is too confusing, that’s perfectly okay. I suggest continuing with the document and revisiting this once you’ve accrued more understanding of pointers.

Strings:

Strings are certainly one of the most confusing aspects of C. This confusion arises because of the fact that there are actually two methods of creating strings, which are oh so very similar, but differ in the minutia, which leads new C programmers to madness. To clarify terminology, a string literal is any string enclosed in double quotes (“”). The confusion already begins here, because string literals are implicitly NULL terminated. The NULL terminator is a special character (the 0th character in the ASCII table), which is escaped by a backslash ‘\0’. This is different from ‘0’. The character ‘0’ is actually decimal 48 in the ASCII table. This distinction is important because it is the reason that our 0s still get printed to the console, meanwhile the NULL-terminator does not. Back to string literals – they are all terminated with the NULL terminator implicitly. The string literal “hello” is actually seen by the compiler as “hello\0”.

Now let’s look at the two methods of creating strings in C. The first method is to create an array of type char. There are several methods of doing this. The following examples all create the string “hello” as an array of type char.

Example 1:

char hello[5];

hello[0] = ‘h’;

hello[1] = ‘e’;

hello[2] = ‘l’;

hello[3] = ‘l’;

hello[4] = ‘0’;

Example 2:

char hello[5] = { ‘h’, ‘e’, ‘l’, ‘l’, ‘o’ };

Example 3;

char hello[5] = “hello”;

Example 4:

char hello[] = “hello”;

Lets look at each of these examples and how they relate, as well as how they differ. First of all, how do these all relate? As mentioned, these are all arrays. Recall that I mentioned in the Arrays section that arrays are contiguous blocks of memory. Using the sizeof() macro, C can correctly read the size in bytes of each of these strings. So where do they differ then? Well, recall that I mentioned string literals are implicitly NULL-terminated in C. Examples 1 and 2 do not use string literals, but examples 3 and 4 do. In the case of example 3, we actually truncate the NULL-terminator, because the size of the array should technically be 6, but we said 5, therefore, it removes the NULL-terminator. If you were to print the sizes of the strings in examples 1, 2, or 3 using sizeof(), you would get 5 as the output. For example 4, however, we get 6 as the output. Since we don’t explicitly state the size of the array in example 4, the compiler reads it for us. The compiler understands that the string literal actually occupies 6 bytes, not 5. For the second type of string that we’ll look at, truncating the NULL terminator is an absolute no-no, and will almost certainly result in the program crashing. For the char array string it is not a problem though, since again, the C compiler knows the size of arrays at compile time. In general, if you’re going to use an array of chars to represent a string, I recommend only using the third or fourth approach. If you are going to use the third approach, however, I recommend making sure that you increase the explicit size of the array to account for the NULL-terminator.

The second type of string that we can create is a char\*. This is, of course, a pointer to a region of memory (can be stack allocated or heap allocated) which stores a seqence of chars (or if it doesn’t, whatever is there will be interpreted as chars). The primary difference between strings created using char\* vs an array of chars is that C does not know the size of the memory region that a pointer points to at compile time. This is why the NULL-terminator is so vital for strings created as char\*s – it indicates where the string stops. Without the NULL-terminator, functions such as printf() will continue to read bytes until they eventually reach the NULL-terminator, which could be several bytes beyond the scope of what you intended. Let’s look at an example that will result in a segmentation fault:

#include <stdio.h>

#include <string.h>

int main(void) {

const char \*a = “hello”;

char \*b = NULL;

strncpy(b, a, strlen(a));

printf(“%s\n”, b);

return 0;

}

This code probably appears pretty reasonable to most novices. We have a string a that we want to copy to b, and then we want to print b to stdout. So what’s the issue? The issue is in the strncpy() function. This function copies up to n bytes from its second argument to the first argument. The strlen() function returns the number of characters in a string *excluding* the NULL-terminator, which is the catch. In other words, strlen(a) returns 5. So we only copy the word “hello” into b, but don’t copy the NULL terminator, which results in accessing memory that is not ours, i.e. a segfault. Note that if we ran the exact same code, but initialized b as char b[5]; the same issue would not arise, even though we’re still omitting the NULL terminator. On another note, the sizeof() operator will always return 8 on 64-bit architecture when trying to get the size of a char\*. As I may or may not have already mentioned, when passing any pointer to sizeof() it always returns the size of the pointer, not the memory that the pointer points to (since C cannot know this at compile-time). If you require the length of the string, you must always do strlen(str) + 1. I recommend using strlen(str) + 1 for both char arrays and char\*s to stay consistent. It gets pretty confusing if you use sizeof() for arrays, but use strlen(str) + 1 for pointers...

One last difference between char arrays and char\*s (though its a bit technical). When we do char \*str = “hello”; you can think of it as if the compiler did something like the following:

const char[] str\_arr = “hello”;

char \*str\_ptr = (char\*)str\_arr;

Normally, C would not allow casting from a const type to a non-const type, but for compatibility reasons, C allows this. This can really screw us up though, since it means that char\*s that point to strings allocated on the stack are actually non-modifiable! For this reason, you should always mark you char\*s as const if they point to stack-allocated memory. If you require a string that is a char\* which must also be mutated, you must do so by allocating memory in the heap. The easiest way to do this is with the strdup() function, which simply calls malloc and returns a heap-allocated copy of the string that you pass it. We’ll look more at strdup() in a bit.

Array of Strings:

Beginners of C often get confused when creating arrays of strings. It is really not that complicated, however. The only way for an array to store multiple strings is to have references to them. So, if we have an array of strings, its type must be char\*. These strings do not have to share the same length since the array is just storing the pointers to each string, which are each 8 bytes. For example, we can create an array of strings using an initializer list like so:

char \*str\_arr[] = {“a”, “ar”, “arr”, “arra”, “array”};

Or, we could create a 2D array like so:

char \*str2D[][2] = {{“NW”, “NE”}, {“SW”, “SE”}};

If we don’t know the values at the time of initialization, that’s fine. We can create a new string using either the array or pointer method and add them just fine.

#include <stdio.h>

#include <stdint.h>

int main(void) {

int i;

char \*arr[6] = {"one", "two", "three", "four"};

/\* Do stuff... \*/

char \*str\_ptr = "five";

char str\_arr[] = "six";

arr[4] = str\_ptr;

arr[5] = str\_arr;

for(i = 0; i < 6; i++)

printf("%s ", arr[i]);

printf("\n");

return 0;

}

// Output:

one two three four five six

String Manipulation:

There are many ways that we can manipulate strings in C. Most of these ways are via functions defined in string.h, which I will cover momentarily, but allow me to explain two useful things that we can do without functions first. These two things apply to both the array and pointer versions of string.

The first thing that you may want to be able to do is extract a particular character from a string. This is most easily done with the index operator ([]). The index operator actually does two things implicitly: It performs pointer arithmetic and then dereferences the pointer/array. In other words, greeting[3] is the same thing as \*(greeting + 3). Since the data type of our string is always char, adding three to greeting means “add 3 \* sizeof(char) to the base address of greeting”. And since sizeof(char) is 1 byte, it would add 0x1 \* 0x3 = 0x3 to the address. Then, by dereferencing at that offset, we return the letter at index 3. So, if our greeting is “hello”, then greeting[3] gives us the char ‘l’ (the second l) and \*(greeting + 3) would give the same thing.

The second manipulation that we can do without functions is offsetting the read position of the string. So, for example, if we only wanted to print the second half of a sentence, we could add the number of characters that we wanted to skip to the base address, and it would only print the string from that offset onwards. This is similar to what we did with the index operator, except that we don’t dereference at the end. Here is some example code demonstrating both of these string manipulations:

#include <stdio.h>

#include <string.h>

int main(void) {

char \*greeting\_ptr = "hello";

char greeting\_arr[] = "hello";

/\* Getting second to last index using index operator \*/

printf("%c\n", greeting\_ptr[strlen(greeting\_ptr) – 1]); /\* Prints o \*/

printf("%c\n", greeting\_arr[strlen(greeting\_arr) - 1]); /\* Prints o \*/

/\* Getting second to last index with pointer arithmetic \*/

printf("%c\n", \*(greeting\_ptr + strlen(greeting\_ptr) - 1)); /\* Prints o \*/

printf("%c\n", \*(greeting\_arr + strlen(greeting\_arr) - 1)); /\* Prints o \*/

/\* Setting read offset for string \*/

printf("%s\n", greeting\_ptr + 1); /\* Prints ello \*/

printf("%s\n", greeting\_arr + 1); /\* Prints ello \*/

return 0;

}

Pointer arithmetic is commonly used to iterate over a string within a loop construct. Since C evaluates all numeric values as true except for 0 (which the NULL terminator happens to be), we can do something like the following:

#include <stdio.h>

#include <string.h>

int main(void) {

char \*str = "faobarbaz";

while (\*str) {

printf("%c", \*str++);

}

putchar('\n');

return 0;

}

We use pointer arithmetic to advance the pointer one char at a time so that \*str evaluates to the next character in the string each time (remember, str++ is the same as str = \*(str + 1);).

string.h:

As I mentioned, there is a header file called string.h which belongs to the C runtime library and includes a few functions that are really helpful for dealing with strings. Here are a couple of the most important ones:

strcpy: char \*strcpy(char \*restrict s1, const char \*restrict s2)

We’ve briefly talked about strcpy. This function copies the string s2 into the string s1. e.g. strcpy(str, “copy me”); Note that the size of s1 must be >= strlen(s2) + 1. The + 1 refers to the NULL character which takes up an additional byte. strcpy clears s1 before copying over s2 to s1. strcpy() is most commonly used for array-style strings, since pointers can simply be reassigned new string literals.

strncpy:char \*strncpy(char \*restrict s1, const char \*restrict s2, size\_t n)

No, I didn’t accidentally write strcpy twice. This function is called str**n**cpy with an ‘n’. Even more confusing is that these functions are very similar, however, it is important that you’re able to distinguish between them! strcpy() will attempt to copy all bytes of the input string, s2, to the output string, s1, including the null terminator. However, strncpy() has a third argument, size\_t n, which specifies how many bytes should get copied, which means that the null terminator might not be copied. If s2 is smaller than s1, the remainder of the space is padded with NULL terminators.

#define STR\_LEN 30

/\* Notice the whitespace left for “jumped”. This will get overwritten \*/

char string[] = “The cow over the moon”;

strncpy(string + strlen(“The cow “), “jumped”, strlen(“jumped”));

printf(“%s\n”, string);

Output: The cow jumped over the moon

strcat: char \*strcat(char \*restrict s1, const char \*restrict s2)

strcat() is used to concatenate/append text to the end of a string. Similar to strcpy, s1 must have enough space left over to be able to insert s2. Unlike strcpy, strcat does not clear s1 before copying s2, however, the pre-existing NULL terminator of s1 gets overwritten by the first character of s2 so that the string doesn’t end prematurely.

strncat:char \*strncat(char \*restrict s1, const char \*restrict s2, size\_t n)

strncat() as you could probably guess, appends the specified number of bytes from the input string, s2, to the output string, s1. Unlike strncpy, strncat does not automatically pad left over space with NULL terminators if s2 is smaller than s1 so you must add a NULL terminator manually.

strlen:size\_t strlen(const char \*s)

Another important one to keep in mind is strlen. An important thing to always keep in mind is that strlen will return the number of characters in the string and will *not* include the null terminator. If you need the size of the string, make sure you do strlen(string) + 1 to account for the null terminator. *Never* use sizeof(string) to count the number of bytes in a string. This was always a point of confusion for myself. Doing sizeof(string) will return the size of the pointer that contains the address of the string literal. Technically doing sizeof(string) when string is an array is acceptable, but it is still preferable to use strlen() + 1 since the size of an array may be larger than the size of the string itself.

strdup: char \*strdup(const char \*s)

We briefly touched on strdup() earlier. This function is quite simple in terms of functionality. In fact, it is more of a helper function, because we could easily write a simple strdup() function ourselves. Here is what that would look like:

char \*strdup(const char \*s) {

char \*ret = malloc(strlen(s) + 1);

if (ret == NULL) return NULL;

strdup(ret, s);

return ret;

}

As you can see, all that strdup() does is allocate some memory on the heap using malloc(), return NULL upon failure, copy the value of the parameter s to our return string and then return the return string.

strtok: char \*strtok(char \*restrict s, const char \*restrict sep)

The strtok() function is akin to split() in most other modern languages. The tok in strtok stands for token. It accepts a token, represented by the sep parameter, and a string that we want to split on, represented by the parameter s. strtok() is a little bit strange due to the fact that it is usually invoked several times. The first time you invoke strtok(), you pass the string you want to split, and the separator token that you want to split that string on (i.e. divide it into multiple sub-strings). For each byte in s, if it matches with a byte in sep, it is replaced by a NULL terminator. This will return a NULL-terminated pointer to the first substring in s delimited by sep. Subsequent calls to strtok() should pass NULL as the argument for s. This will cause strtok() to continue returning pointers to the next substring in s delimited by sep. When strtok() runs out of substrings (or if there were none to begin with), it returns NULL. Here is a typical use-case for strtok():

int main(void) {

const char \*line = "line to be separated";

char \*lptr = strdup(line);

const char \*token = " \n\t";

char \*next = strtok(lptr, token);

do {

printf("%s\n", next);

} while ((next = strtok(NULL, token)) != NULL);

free(lptr);

return 0;

}

A few things to note about this code. First of all, it’s bad practice for me to be using a do-while loop here since next could easily contain NULL if the separator did not exist within line. I happen to know by looking at line that it contains spaces, so I know that printf() will succeed for the “do” part of the do-while, but this is not a good assumption to make in real code. Second thing to note is that I’m using strdup() here to return a pointer to a heap-allocated copy of line, which I store in lptr so that it can later be freed. If I just passed in line by itself to strtok(), I would first get a compiler error because line is const. If I removed the const modifier, however, we’d get a segfault, because as I mentioned earlier, line points to a stack-allocated string which is read-only (however the compiler allows the implicit cast to succeed). Note that instead of using a char\* and strdup(), we could’ve also just used a non-const char array, which would save us from having to call free() later on. The last thing to note is the string placed in the token variable. I didn’t really cover this in detail in my explanation since I figured it’d be easier to simply demonstrate. Some languages allow the delimiter to be an entire substring, but strtok only allows for individual characters. The sep parameter is still a string, however, because we can specify multiple characters to separate on. In this case, all spaces, newlines, and tab characters are replaced with a NULL-terminator by strtok().

strstr: char \*strstr(const char \*s1, const char \*s2);

strstr() checks if a substring s1 exists within the string s2. If s1 does exist within s2, the function returns a pointer to the starting location of s1 within s2. If s1 does not exist within s2, the function returns NULL.

Allocating Memory:

One of the most powerful things about C is that is allows us to allocate our own memory. In most higher-level programming languages, memory is allocated dynamically. C will automatically allocate memory for us as well. For example, you don’t have to manually allocate memory for an int because C does this for you. C allocates local variables. A local variable is one which belongs within some scope (curly braces). This region of memory is known as the stack, and we’ve covered it briefly before. Your program has multiple stack frames, each with their own space reserved for local variables. The issue is when we need to reference a variable that is does not belong to the same scope. This is where pointers come in handy. A pointer can point to a local variable (which is what we’ve covered thus far), but it can also point to a general-purpose region or block of memory which is allocated in heap memory. Heap memory, or simply “the heap” for short, is a region of memory on your system that can be used for general purpose allocations. Any memory that is allocated from the heap remains allocated until either free() is called, or the process terminates and the OS retrieves all of its resources (including heap allocated memory). In general, the heap has very slow access times, and should be avoided where possible. It is imperative that anytime you are no longer using memory allocated in heap space, you call free() to release it. Failing to do so can lead to memory leaks, which is when your program creates a pointer to some memory on the heap, but then forgets to free it. Once the pointer to that memory block is destroyed (from going out of scope i.e. the stack frame getting popped off) then that memory will remain in the heap untouched until the process terminates. Using the heap is often useful for large allocations (like a buffer), or any variable that must remain in memory beyond its local scope. Here are a few ways to allocate memory:

alloca: void \*alloca(size\_t size)

alloca() is sort of an underrated function in many ways. It typically gets overshadowed by the next function we’ll look at, malloc(). The difference between malloc() and alloca() is that malloc() returns a pointer to memory allocated in the heap, whereas alloca() returns a pointer to memory allocated on the stack(). The most effective use for alloca() is creating pseudo arrays with variable length. alloca() will return a pointer to a memory segment size bytes long, and we can use pointer arithmetic to index through the “array”. For example: uint16\_t \*array = alloca(10 \* sizeof(uint16\_t)); This will allocate 20 bytes on the stack that we can index through using array++ or array-- and then dereferencing whenever appropriate. Since alloca() allocates on the stack, it is not necessary to call free(), unlike with malloc() or calloc(). Note that alloca() is not a POSIX function, which is arguably its greatest misfortune, and likely the reason you don’t see it used very often.

malloc: void \*malloc(size\_t size)

malloc() is the most common method of allocating memory on the heap. Much like alloca(), it takes in one parameter: the number of bytes to allocate on the heap. malloc() returns a void pointer which contains the address to the start of the allocated memory on success, or it will return NULL on failure. Also note that memory which is requested by malloc is not backed right away by the MMU. For instance, if you requested 15GB of data to be allocated on the heap, malloc would accept your request, but the virtual pages would only get backed to PAS at the time of writing. This is called Copy on Write (COW). Here is a common use-case of malloc():

#include <stdio.h>

#include <stdlib.h>

int main(void) {

size\_t buf\_size = 4 \* 1024; /\* Allocate 4KB \*/

char \*char\_buf = malloc(sizeof(char) \* buf\_size);

if (char\_buf == NULL) {

printf(“Malloc failed to allocate memory\n”);

return -1;

}

/\* Do stuff ... \*/

}

malloc() returns a void pointer which contains the address to the start of the allocated memory on success, or it will return NULL on failure.

calloc: void \*calloc(size\_t nelem, size\_t elsize)

calloc() is similar to malloc(), although calloc() will initialize all bits of the memory to 0. You’ll notice that calloc() takes 2 parameters instead of one. The first is the number of elements, and the second is the element size. Typically, this is not that useful – the number of elements is usually just one, which is multiplied by the element size to get the final number of bytes to allocate. The nelem parameter may be useful for arrays if you give it the number of elements in the array, and then elsize can be sizeof(data\_type). malloc() does not initialize the memory it returns, so it will contain garbage, therefore calloc() is useful if you need extra safety. calloc() also returns NULL on failure.

realloc: void \*realloc(void \*ptr, size\_t size)

realloc() is used to re-allocate memory to a new size. Typically, it is used to dynamically increase the size of an object, however, it could technically be used to shrink it as well (although this can be dangerous if you still need to preserve the contents of memory). realloc() simply takes in a pointer to the allocated memory as it’s first argument, and then the size parameter determines the new size of the memory pool. Upon failure, NULL is returned.

Memory Manipulation:

There are a few POSIX functions in the C runtime library that are useful for manipulating memory. These functions are most commonly used on heap memory (e.g. after a call to malloc() or calloc()), but work equally fine on memory that has been allocated locally on the stack.

memset: void \*memset(void \*s, int c, size\_t n)

memset() is a useful function for reinitializing memory that has already been allocated. memset() allows us to initialize each byte in the memory block with an integer value of our choosing. For example, we could use memset to initialize each byte to contain the number 5. Allocating memory on the heap using malloc(sizeof(data)), followed by memset(\*data, 0, sizeof(data)) is equivalent to doing calloc(1, sizeof(data)). The first parameter is an optional pointer. This is used if you are overwriting an existing pool of memory. If set to NULL, memset() will pick an address for you. The second parameter, c, is the character that you will be writing to each byte. Technically you’re meant to input an int for this parameter, but it gets cast to an unsigned char internally. Finally, n determines the number of bytes that memset will copy c into. memset() does not return an exit code.

memcpy: void \*memcpy(void \*restrict s1, const void \*restrict s2, size\_t n)

memcpy() performs a deep copy of the object pointed to by s2 into the object pointed to by s1. If memcpy() copies over two regions of memory that overlap, it leads to undefined behavior, so it can be somewhat dangerous to use. For example, if the last byte of s2 is the same byte in memory as the first byte of s1, this can lead to subtle bugs. This is because memcpy uses restrict on its pointers which allows the compiler to make optimizations under the assumption that the memory pointed to by s1 does not overlap with the memory pointed to by s2. memcpy() returns a pointer to s1, and nothing is set to indicate errors (so use with caution).

memmove: void \*memmove(void \*s1, const void \*s2, size\_t n)

memmove() is essentially identical to memcpy(), except that memmove() does not use the restrict keyword, meaning that it allows for transfers between overlapping data. It can do this safely by first copying the source data (s2) into a temporary buffer that does not overlap with either s1 or s2 before it gets copied again into the destination buffer (s1). Due to the intermediary copy, memmove() is safer than memcpy() at this cost of being less efficient. There is much debate over whether to use memcpy() or memmove(). I feel that it really depends on the context, but in general, if your memory is overlapping its likely because you’ve intentionally created a pointer that points to memory already pointed to by another pointer. This is called double-ownership and is generally a bad practice in programming. Instead, you should think about refactoring your code so that each pointer has exclusive access to the memory that it points to.

memcmp: int memcmp(const void \*s1, const void \*s2, size\_t n)

We’ve already looked at strcmp() which is used to check if two strings are the same in C. This is, of course, a wrapper around the memcmp() function, which does a byte by byte comparison up to n bytes between two values of any type. The function will either return 0, a negative value, or a postive value, to indicate whether the object pointed to by s1 is the same, less than, or more than the value pointed to by s2 respectively.

Structs:

Structs, short for “structures”, are a very important aspect of C programming (and for that reason, we will likely be spending a good amount of time on them). New programmers coming from OOP languages are usually quick to point out the similarities between structs and objects, however, they should not be compared for multiple reasons. While both objects and structs encapsulate data, that is pretty much where their similarities end. Structs cannot implement or inherit from other classes (no polymorphism) and they cannot create instances of classes which invoke methods (no instantiation), meaning that structs share none of the core properties of an object in OOP. A struct is more akin to an array that is capable of storing multiple data types. They are used when we want to create a packet of logically related data that will typically be accessed multiple times. Think of them as arrays containing an assortment of variables of differing data types. With the description out of the way, lets look at how we can create a struct.

Creating a Struct:

In order to create a struct, we use the keyword “struct” followed by the name of the struct, and then we encapsulate its members within curly braces. These curly braces end in a semi colon since a struct is technically all one declaration, however, we often split that declaration up into individual lines for readability. Here is an example of creating a struct called book:

struct book {

int pages;

char title[50];

char author[50];

bool soft\_back;

};

Note that we never initialize the members of a struct within the declaration. This is meant to be done when we create a new instance of struct. In order to create an instance of struct, we first specify that we are using the struct that we’ve created, followed by the name of the instance like so: struct book newbook; This will create an instance of book called newbook. Notice that we have to specify that book is a struct, otherwise C would not know what we were referring to. We will see why this is in a bit. Book has three members: pages, author, and soft\_back. Assuming that the members have a storage class of automatic, these will contain garbage values by default. In order to initialize everything to 0 in a struct, we can use an empty initializer list like so: struct book newbook = { 0 }; In reality, this pnly tells the compiler to set the *first* member of the struct to 0, however, in C if we initialize only the first ‘n’ elements of a struct using an initializer list, the rest get initialized to 0. So in effect, setting newbook = { 0 }; actually initializes all members of newbook. Alternatively, we can actually populate these values like we can with an array if we know what we want to set the members to on creation. For example:

struct book newbook = { 927, “Moby Dick”, “Richard Bently”, false };

There is also an another method of using initializer lists when it comes to structs which is a bit more readable. Usually structs are declared in header files, so with the following syntax, we can get a clear idea of which members are being set:

struct book newbook = {

.pages = 927,

.title = “Moby Dick”,

.author = “Richard Bently”

};

As I mentioned before, any members which aren’t explicitly set will be initialized to 0.

This is great if we know what we want the members to be at the time of initialization, but if we need to read or write to the structs members after that, then we need to know how to access those members. This is typically done using the dot operator. For example, if we wanted to print each member of the previous struct:

printf(“%d\n”, newbook.pages);

printf(“%s\n”, newbook.title);

printf(“%s\n”, newbook.author);

printf(“%s\n”, (newbook.soft\_back) ? “true” : “false”);

The dot operator can also be used to update members of the struct. These are the basics of using structs. Now we will look into some more complex aspects of structs.

Struct Size:

Often times we need to know how big a struct is. This should always, always, always be calculated using sizeof(struct), but I am going to explain to you exactly how we *could* calculate the size of a struct without using sizeof(), and *why* we should always use sizeof(). As a very basic explanation, we can say that the size of a struct is the sum of each of its member’s data types. In the case of book, we can guess that its size will be 105 bytes, since int takes up 4 bytes on a 64-bit architecture, bool takes up 1 byte, and each array takes up 50 bytes. That would give us a total of 105 bytes. However, if we do a printf like so:

printf(“Sizeof book %d\n”, sizeof(struct book));

Then we will get back 108. This is due to a phenomenon called structure alignment. Structure alignment is done by the compiler to optimize bus transactions between the CPU and memory. Because the bus size is dependent on the architecture of the CPU, transaction across the bus are much faster if they are some multiple of the word size. This is better demonstrated using the following two structs:

struct test1

{

short s; /\* [////] [////] [ ] [ ] \*/

int i; /\* [////] [////] [////] [////] \*/

char c; /\* [////] [ ] [ ] [ ] \*/

};

struct test2

{

int i; /\* [////] [////] [////] [////] \*/

char c; /\* [////] [ ] [ ] [ ] \*/

short s; /\* [////] [////] [ ] [ ] \*/

};

Both structs test1 and test2 contain the same variables, and yet, if we print their sizes using sizeof(), test1 measures 12 bytes, and test2 measures 8 bytes??? Why is this? Well, as I alluded to, the C compiler will try to optimize by aligning the bytes of the structs members if possible. Since the word size on a 64-bit architecture is 4 bytes, the compiler tries to pack members into packets of 4 bytes. This happens from top to bottom, so in the case of test1, the compiler is not able to align any members because trying to pack short s and int i wouldn’t work (since i takes up a full 4 bytes), and same goes for int i and char c. In the case of test2, this actually is possible. We cannot combine int i and char c since int i takes up a full 4 bytes, however, char c and short s can combine into 1 packet to conserve space like so:

char c short s empty

[/////////] [/////////] [/////////] [ ]

In conclusion, it actually matters where we place our members within structs because if we place them in a poor configuration, we are wasting free space. Try to group shorts together since two shorts combine to make 4 bytes, and then any 1 byte data types can be grouped together as well. As a good programming practice, it is good to add an array at the end of your structs called “padding” which takes up the remaining space that goes unused. For example, take the following struct:

struct example {

int i1;

int i2;

short s1;

short s2;

char c1;

char c1;

char padding[2]; // Padding of 2 bytes for struct alignment

};

Adding a padding variable is nice because you are losing that space either way. At least this way, it is possible to use padding for something if you decide that it could be beneficial to you, and it helps show your colleagues that you’ve put struct alignment into consideration and have calculated the optimal layout.

Bit Fields:

Oh wait, you thought we were done with struct alignment? Unfortunately not. Bit fields take struct alignment to the extreme, not just down to 4 bytes alignments, but down to packing individual bits. The concept here is simple but it can be difficult to know when it is and isn’t a good idea to use bit fields (the answer is not often). In order to use bit fields, we put a colon after the member and the number of bits that we want it to occupy. For example, we could create the following struct:

struct bitfields {

unsigned boolean : 1;

unsigned octal : 3;

char nibble : 4;

char padding[3];

};

In the example above, we use bitfields to reduce the size of the members. Don’t be fooled here, there is no “unsigned boolean” data type here. unsigned int can be shortened to just “unsigned” and then boolean is the name I gave to my own definition of boolean. Since booleans are represented as either 0 or 1, I made this member occupy just one bit. The next member I created is also an unsigned int with only 3 bits, which I called octal since we can represent octal numbers with 3 bits. Next, I created a char called nibble which occupies 4 bits. It doesn’t really matter what the data type is in this context. Since int and char are both numeric types, and since we are defining how many bits we are using anyways, it doesn’t really matter whether we use int or char. Since these 3 members accumulate into 1 byte, and our struct alignment packs its transactions into packets of 4 bytes, I’ve added an array of size 3 to pad this struct out. Hopefully that should give you a good enough explanation of how to use bit fields. These should probably be used with caution, since performing arithmetic with strange bit lengths can probably result in unexpected behaviour, and you will lose a lot of accuracy.

Here is the only real life situation I’ve come across personally where I felt that bit fields were a good decision. Imagine you are trying to virtualize/emulate a microprocessor, and a couple of the registers have an odd number of bits in them. In this hypothetical, lets say that we have a register ‘z’, which is 9 bits. The z register can be accessed in one of two ways: you can either read the whole 9 bits, or more likely, you just want to read the least significant byte of those 9 bits. Let’s call the full 9 bits z, and the LSB zl. In order to program this in code, we can use a combination of bit fields and unions. Bit fields can only take away bits, not add them. We’ll use a 16 bit data type such as an unsigned short to represent z, and we’ll use an unsigned char to represent zl. This might look something like the following:

struct processor {

uint16\_t z : 9;

uint8\_t zl;

};

Accessing zl of the processor struct effectively masks out the upper bit of z, whereas z accesses the full 9 bits. Another benefit of doing it this way is that a good compiler will be able to tell you if an overflow occurs in z accounting for the fact that it only has 9 bits. For example, trying to assign z with a value of 0xFFFF will result in a warning: “Implicit truncation from ‘int’ to bit-field changes value from 65535 to 511”.

Pointers to Structs:

We can create pointers to structs, naturally. Similar to arrays, structs have a base pointer, so we can create pointers to structs. This works the same way any other pointer would work. First, assume that we have a struct called vehicle. We will declare an instance of vehicle called car like so: struct vehicle car = { 0 }; Then, we can create a pointer to car like so: struct vehicle \*pcar = &car; But now, we face a problem. In ord1er to set the members of car, we can no longer use the dot operator. In order to solve this, we will need to use a different operator called the arrow operator which looks like this “->”. The arrow operator does the same thing as the dot operator, but it also dereferences the memory that it accesses. For example, let’s say that the vehicle struct has a member int wheels, which describes how many wheels the vehicle has. In order to access wheels using pcar, we could either do \*(pcar.wheels) or we could do pcar->wheels. Both of these statements are equivalent.

Nested Structs and Anonymous Inner Structs:

We can create structs within structs if we are so inclined, which is called “nesting structs”. Inner structs can either be pre-existing struct types, or we can optionally define an inner struct within the parent struct. Here is an example of the former method of nesting:

struct occupation {

char job\_title[50];

int years\_worked;

};

struct person {

char name[30];

struct occupation job;

int age;

};

In this example, the struct occupation is created, and then we nest it within another struct called person. In this case, the person struct will always have an occupation field, but we can also just create an occupation object by itself if we so choose. Now let’s look at an example of the latter method of nesting, where we declare a new struct within a struct:

int main (void) {

struct person {

char name[30];

struct occupation {

char job\_title[50];

int years\_worked;

} job;

int age;

};

struct person john = {

.name = "John Doe",

.job.job\_title = "Software Engineer",

.job.years\_worked = 5,

.age = 26

};

printf("Occupation of %s : %s\n", john.name, john.job.job\_title);

printf("Years worked : %d\n", john.job.years\_worked);

return 0;

}

This code might be a little bit confusing, but we’ll take it step by step. In main, we create the same struct as before, person. Person has their name member and age member as usual, however, now we’ve included their occupation as an inner struct. The important thing to note here is that we create an instance of occupation called job between the closing curly brace and the semi-colon. This is a valid thing that we can do with structs. We are allowed to pre-define certain objects when we create the struct so that they can immediately be used afterwards. If we want to create multiple objects, then they should be comma separated. Notice that we use the dot operator to access job, and then we use the dot operator again to access the members of job.

The final way of nesting structs is to create an anonymous inner struct, which were introduced in C11. This is when we create a struct type within a struct, but we don’t give the struct a name. The actual practicality of this is beyond the scope that I’m comfortable explaining, but this is becoming a more common practice. Here is a code example demonstrating an anonymous inner struct:

struct color {

struct {

int r, g, b;

};

float a;

} red, orange, yellow, green, blue, purple, magenta;

int main(void)

{

red.r = 255;

red.g = 0;

red.b = 0;

red.a = 1.0f;

return 0;

}

Here, in order to access the members r, g, and b, we simply pretend as if the members of the anonymous struct were part of the parent struct to begin with. If we had given the anonymous struct a name (eg. struct rgb), then we would’ve had to access those members like so: red.rgb.r = 255; As I mentioned, why this is useful you can research on your own, as it is beyond the scope of this document.

Self-Referencial Structs:

Another possibility using structs, is to create self-referencial structs i.e. structs which contain their enclosing type as a member. This is often used in linked lists for instance, where we need to keep track of the next and/or previous nodes in the list. For example:

struct Node {

int data;

node prev;

node next;

} \*node, \*tail, \*head;

Here, note that both prev and next are of type node (which is actually struct Node\*).

Struct Namespaces:

There are multiple ways to declare structs. Each method alters the scope (visibility) and capabilities of the struct. C distinguishes between the variable/function namespace and the struct namespace. For this reason, it is valid to have a struct and a variable or method that share the same name. It should be noted that typdef is commonly used on structs and for good reason. Using typedef, we can turn instances of a struct into aliases so that we no longer have to type out struct <struct\_name> each time that we would like to create a new struct. Here are the possible combinations of struct declaration:  
  
**1. No name or instance(s):**

struct {

int member1;

char member2;

};

As we seen, this is an anonymous struct. No name gets added to the struct namespace and it may only be referenced in the context of being nested within another struct.

**2. With name but no instance(s):**

struct Foo {

int member1;

char member2;

};

Using this method, struct Foo gets added to the struct namespace. Since we have not pre-defined any instances of Foo, we must declare a new struct of type Foo like so: struct Foo foo;

**3. With name and instance(s):**

struct Foo {

int member1;

char member2;

} foo;

This case is the same as the last with the exception of pre-defined instances of Foo. In this case we create one such instance called foo so that we can begin using that immediately. This will add both Foo and foo to the struct namespace.

**4. With no name, instance(s), and typedef:**

typedef struct {

int member1;

char member2;

} foo\_t;

Here we may need a refresher on typedef. The important thing to remember is that the typedef name is an alias for the type that preceeds it. In this case, the instance’s name is also treated as the typedef name. Rather than adding foo\_t to the struct namespace, it gets added to the variable/function namespace, so creating a variable or method name called foo\_t (in this example) will throw a compile-time error. foo\_t will no longer be an instance of Foo, but rather an alias as I mentioned. So in order to create an instance of foo, we must now say foo\_t foo;

**5. With name, instance(s), and typedef:**

typedef struct Foo {

int member1;

char member2;

} foo\_t;

In this final case, we get the best of both worlds. This adds Foo to the struct namespace, and foo\_t to the variable/function namespace. Now we can either declare a new Foo like so: struct Foo foo; or like so: foo\_t foo; This is the most commonly used way to create a struct.

I understand that it may seem a daunting task to memorize all of these various combinations, but once you get a grasp on structs and namespaces, it becomes more clear as to the reasoning of why certain declarations do or do not work.

Unions:

Unions are very similar to structs but have one key difference: Structs allocate space for each member, but unions only allocate space for one member. This means that a union is only ever as big as its *largest* member. The memory that the union allocates is cast to whatever the data type is of the data that we are currently trying to access. We need to be very careful with unions because we can only ever access one member at a time. This is because when we access a member of a union, it clobbers the member which was last accessed with the new variable’s data. Unions are typically used when we want to choose between one of many related “things”. A union can switch between those different things on the fly or simply pick one and stick with it.

The following code demonstrates the incorrect way of accessing data using a union:

#include <stdio.h>

#include <string.h>

int main (void) {

union test {

int i;

float f;

char s[20];

};

union test error;

error.i = 10;

error.f = 33.3333f;

strcpy(error.s, "error");

printf( "error.i : %d\n", error.i);

printf( "error.f : %f\n", error.f);

printf( "error.s : %s\n", error.s);

return 0;

}

// Output:

error.i : 1869771365

error.f : 75033666879216344191480102912.000000

error.s : error

The reason it prints all this garbage is because we try to access each member of the union, but the union currently holds the data for s, and nothing else. That is because s was the last value that we wrote to (and therefore, the string literal “error” is still present). In order for us to print out each member, we would have to break up the print statements like in the following code:

int main (void) {

union test {

int i;

float f;

char s[20];

};

union test error;

error.i = 10;

printf( "error.i : %d\n", error.i);

error.f = 33.3333f;

printf( "error.f : %f\n", error.f);

strcpy(error.s, "error");

printf( "error.s : %s\n", error.s);

return 0;

}

// Output:

error.i : 10

error.f : 33.333302

error.s : error

Error Handling:

Error handling in C is a bit odd, and probably underwhelming if you’re coming from higher level languages with built-in exception handling and all that stuff. At the root of it, C really has no concept of “errors”, and it really gives us no built-in tools to help us handle errors when they occur. The solution for years has simply been to return an integer that we call an error code depending on what happened, and then we can decide what to do whatever we wish with that. A value which is associated with a special meaning or significance can also be referred to as a sentinal value. For example, returning 0 to indicate that there were no errors means that the number 0 could be considered as a sentinal value. I will be going over macros first, and then functions that we can use to help us. Note: you will need to include <errno.h> to use most of these.

Error Macros:

**errno:** Likely the most important macro for error handling is errno. This integer is unique because its value changes depending on the return value of the previous function call. There are also special functions that we will go over momentarily for printing out the textual representation of errno. Many of the C runtime library functions will set errno to some macro defined by POSIX. These include ENOMEM, EAGAIN, EBADF, EBUSY, etc. just to name a few. For the full list of error macros, see the man pages for errno.h. It is best practice that you do not modify the value of errno.

**EXIT\_SUCCESS:** This macro was created with the intent of being used as an argument to exit() (but could just as easily be used for any return statement). EXIT\_SUCCESS is defined as 0 and is primarily used for increased readability. I personally recommend only using it for calls to exit() or the return statement for main() since it implies that the calling thread is terminating.

**EXIT\_FAILURE:** By default set to 1, EXIT\_FAILURE is the counterpart for EXIT\_SUCCESS. This is used to indicate that something fatal happened during the execution of the program causing it to terminate the calling thread.

Error Functions:

**perror:** void perror(const char \*s)

perror() is a function that is really centered around the errno macro. You supply perror with your own message, and then perror() appends a colon, a space, and the textual representation of the current value of errno. Obviously the textual strings correspond to the error macros defined by POSIX. In other words, you cannot just set errno to some random number and expect perror() to print out the error message that you expect. Note that the output of perror() will get printed to stderr.

strerror: char \*strerror(int errnum)

strerror() is very similar to perror() but it only returns the textual representation of errno (nothing more). You can achieve the exact same effect of calling perror() by calling fprintf() (like printf but prints to the specified file stream) like so: fprintf(stderr, “Error opening file %s\n”, strerror(errno));

assert: void assert(scalar expression)

assert() is not really considered “error handling”, but it can help us avoid nasty errors. You provide assert with an expression to evaluate as you would with any conditional loop. Upon success, the program continues, but if the expression evaluates to be false, information about the assert call is written to stderr and the program calls abort(), taking your program down with it. assert() is a macro, and it good to use in areas where you know for certain that an expression should evaluate to true.

static\_assert: void static\_assert(scalar expression)

static\_assert() functions identically to assert() with the key destinction being that assert() is evaluated at runtime, and static\_assert() is evaluated at compile time. This is not as useful as assert() because in most cases our variables change over the course of the program and that is something only assert() can catch. Still, static\_assert() can be useful for debugging or making sure someone doesn’t make a common mistake in the code.

exit:void exit(int status)

Once again, not really an error handling method, but exit() is often called when we want to terminate the program (either because it has finished serving its purpose or because we encountered an error that could not be handled). A good example of when to use exit() would be when opening a file. Assuming that the file is required for the program to continue, then perhaps it is best to print a message telling the user to supply the file, and then exiting. exit() is the cleanest way of ending the program because it closes file descriptors, flushes buffers, and deletes temporary files.

Variable Arguments:

You may or may not be familiar with variable arguments (a.k.a. variadic arguments) if you have some degree of programming experience. If you’ve made it this far, you’ve actually seen variable arguments being used before, but may not have known it. The term variable argument (var-arg for short) is used to suggest that the number of arguments that we pass to a function can vary in number. printf() and scanf are common examples of this, because we can pass a varying number of arguments either after the format string. Var-args require a bit more setup in C than in most other languages, unfortunately. As is the case with most other languages, var-args are denoted using elipses (...). These elipses must be placed as the last argument within a function’s signature, after all other parameters. For instance, we could have a function called fmtstr() to format our strings. As the first argument, we could take a format string, followed by a variable list of data which corresponds to the string:

char \*fmtstr(const char \*fmt, ...);

In order to use var-args, we need to include <stdarg.h>. This has a few macros that we can use here for creating a list to contain our var-args. First, we declare a va\_list object like so:

va\_list args;

This will create a pointer to the start of the arguments list. Since each argument within the var-args list can be its own distinct type, the va\_list must be of type void \*. Next, we initialize the list. We do this by supplying the va\_start() macro with the va\_list object, followed by the last known fixed argument of the enclosing function. Following the fmtstr() example I provided, this would look like the following:

va\_start(args, fmt);

Any function which accepts variable arguments *must* accept at least one argument aside from the arguments themselves. I’m not familiar with any good ways of abstracting this away in the case that you don’t require the first argument aside from complex macros that would not be worth the hassle. After initialization, we can extract the arguments sequentially using the va\_arg() macro to retrieve the next argument in the list. The va\_arg() macro accepts the va\_list as its first parameter, and the type that we want to cast to as the second parameter:

va\_arg(args, const char \*);

The va\_start() macro appends a NULL terminator to the end of the list for us, so in order to iterate over each argument we can simple use a while loop that checks if va\_arg() returned NULL. Finally, we deallocate the va\_list using va\_end(), which just takes the va\_list object as its only parameter:

va\_end(args);

The only other macro in the stdarg.h header is va\_copy(), which can duplicate a va\_list. It takes in two parameters: dest and src. It will copy the contents from src to dest.

I/O:

We will now be looking at a very large section dedicated to I/O. Some of this will be related to user I/O through the command line, and some will be through files. I/O is a very important part of C because reading and writing are truly the only two commands that we can do on a computer to make things happen! I will preface by saying that user input is handled quite poorly in C, as it was not really the focal point of the language. We will look at some best practices and methods for dealing with user input regardless.

printf and scanf:

printf() stands for print and format. printf is the most common way that you will see for outputting text to stdout. scanf() is one of several ways to accept user input, and works similarily to printf(). Both use format specifiers to tell the functions how to interpret incoming or outgoing data. We will begin by going into detail on printf() and then moving on to scanf().

printf: int printf(const char \*format, ...)

printf is the standard way for printing text to the console via stdout. The file stream cannot be changed to something other than stdout using printf(). The first argument is the format string which is essentially our output text. This string can include special characters called format specifiers which are listed below, and will be used by printf() to determine how to interpret the outgoing data. The “second” parameter of printf() are the variadic arguments which we discussed earlier. This means that you can give as many arguments to printf as you want, so long as the first argument is the format string. The subsequent arguments must align with the format specifiers that you give in the format string. So for instance, if %d is the first format specifier in your format string, then the first variadic argument that you give to printf will be interpreted as a decimal. Here are a list of format specifiers and what each of them mean to printf:

**Format Specifier: Type:**

%c Char

%d Signed int

%e | %E Scientific notation of float

%f Float

%g | %G Same as %e | %E

%hi Signed short

%hu Unsigned short

%i Unsigned int

%l | %ld | %li Long

%lf Double

%Lf Long double

%lu Unsigned int or unsigned long

%lli | %lld Long long

%o Octal

%p Pointer

%s String

%x | %X Hexadecimal (lowercase or uppercase)

%n Nothing

%% Prints ‘%’

Note that we can also pad strings by placing a number in between the the format specifier. Here is an example with a string:  
int main(void) {

printf("|%10s|\n", "hello");

return 0;

}

This will print:  
| hello|

Alternatively, we can make the padding length negative to have the padding on the right-hand side, rather than the left, which is the default:

printf(“%s-10s\n”, “hello”);

Prints:  
|hello |

I find that the best usecase for the padding feature in printf is when you are trying to output a sequence of bytes in hexadecimal format. The %x or %X format specifiers allow us to add the number of padding spaces that we want, as well as an optional number that will replace the default use of spaces for the padding characters.   
  
#define BUF\_SZ 2

int main(void) {

int i;

uint8\_t bytes[BUF\_SZ] = { 0x0B, 0xFF };

for (i = 0; i < BUF\_SZ; ++i) {

printf("0x%02X\n", bytes[i]);

}

return 0;

}

Prints:  
0x0B

0xFF

Finally, I want to briefly touch upon the PRI macros introduced in C99 within the inttypes.h header file. PRI macros ensure that the correct format specifier is selected, regardless of platform, when using certain types, such as those in the stdint.h header file. For example, when printing a uint32\_t, we should be using the equivallent PRIu32 macro to ensure that our formatting will be cross-platform. Here is an example:  
  
#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

int main(void) {

uint32\_t num = 32;

printf(“%” PRIu32 “\n”, num);

return 0;

}

Since the PRI macros are string literals, we can easily integrate them into our format string due to the fact that C allows for automatic string concatonation when multiple string literals are separated by white space. Many other PRI macros exist, so I will refer you to read the man pages for inttypes.h if you’d like to know more.

scanf: int scanf(const char \*format, ...)

scanf tries to immitate printf() in its function signature. As you can see, scanf() also accepts a format string, but rather than formatting outgoing text, we provide scanf() with format specifiers to indicate the data type of incoming data. scanf() will await input from the user on the command line. When the user presses enter, scanf() reads from the input buffer and interprets each whitespace-separated word by its corresponding format specifier. It then places the interpreted text into the corresponding variable that was provided in the var args list. When providing variables with scanf, you must provide their addresses using the address operator. This is because scanf will be writing to these variables so we must pass by reference, not value. We also should not add any text other than spaces and format specifiers within the format string. scanf() is considered to be dangerous for many reasons. The first is that we cannot guarantee that the user will follow the format string. scanf() also doesn’t flush the buffer, so it will leave the newline character, or any other data that was not read, sitting in there until it gets read by accident. We also can’t guarantee that the values entered through the command line will fit the boundaries of our variables, and there is no way that we can do error checking before the user input gets placed into the variables by scanf(). For all these reasons and more, scanf() should be avoided in 95% of situations.

Other Functions for Standard I/O:

C has many other functions for receiving and outputting text to the console. This can get quite confusing, as many of these are becoming deprecated due to vulnerabilities with buffer overflow/stack smashing exploits. Most of these functions are similar, but have a few key differences that may make them more or less suitable depending on the given situation. Here is a list of the ones that we will cover before going over each individually:

Standard I/O functions

* getc(FILE \*stream)
* getchar(void)
* putc(int c, FILE \*stream)
* putchar(int c)
* gets(char \*s)
* puts(const char \*s)

File I/O functions

* fgetc(FILE \*stream);
* fputc(int c, FILE \*stream)
* fgets(char \*restrict s, int n, FILE \*restrict stream)
* fputs(const char \*restrict s, FILE \*restrict stream)
* fread(void \*restrict ptr, size\_t size, size\_t nitems, FILE \*restrict stream)
* fwrite(const void \*restrict ptr, size\_t size, size\_t nitems, FILE \*restrict stream)

getc: int getc(FILE \*stream)

The getc() function is the standard way of receiving a character from a specified file stream. As you may know if you read my document on GNU/Linux, there are 3 primary I/O streams: stdin, stdout, and stderr. These are defined by the system, so stdout for example, may not point to the same location on all systems. Usually though, these will point to your terminal. Typically, you would use getc() to read the first character from stdin (the input buffer of the console/terminal). E.g. getc(stdin); would wait for the user to enter in a single character before proceeding. getc() returns the integer representation of the last character that was read.

getchar: int getchar(void)

You’ll notice that getchar() does not accept any parameters. This is because getchar() is equivallent to getc(), except when it comes to the stream. getchar() only reads from stdin, and therefore, is equivallent to getc(stdin).

putc: int putc(int c, FILE \*stream)

The putc() function is very similar to getc(), with the difference being that putc() is for writing a character to a stream, whilst getc() is for reading. The output stream is typically either stdout or stderr, although you can write to any file stream you wish.

putchar: int putchar(void)

Hopefully you’ve been able to deduce that if printf() and scanf() are a pair, as are getc() and putc(), therefore getchar() and putchar() must also be a pair. If you arrived at this conclusion, congratulations, you are correct! putchar() is indeed equivalent to putc(stdout).

gets: char \*gets(char \*s)

The gets() function is one of the few I/O functions that is **deprecated**. We will still cover it regardless. gets() was used for receiving entire strings from stdin, not just characters. Note that there is no parameter for a file stream. This is because gets() assumes stdin as the file stream. It continues to read characters until either a newline feed is read (‘\n’), or until it reads EOF, which is just a macro set to 0, but used to indicate the “End of File”. gets() writes each character into the output buffer s. This function is deprecated because it provides no protection against buffer overflow i.e. reading too many characters into the output buffer s, so that it overflows and causes memory corruption.

puts: int puts(const char \*s)

Like gets(), the s in puts stands for “string”, as in, put the given string s into stdout. Unlike gets(), puts() is not deprecated.

fgetc: int fgetc(FILE \*stream)

The fgetc() function is pretty much identical to getc(), except for the fact that getc() can be implemented as a macro, whereas fgetc() cannot be implemented as a macro. This is a pretty trivial difference, but in essence, fgetc() is sort of useless since it operates slightly slower than getc().

fputc: int fputc(int c, FILE \*stream)

fputc() shares the same difference with putc() that getc() and fgetc() share. Even though putc() can lead to undefined behaviour if implemented as a macro (which you probably shouldn’t do by the way), the slight reduction in efficiency of fputc() makes it rather obsolete. putc() works more than fine, so fputc() does not have a very good use-case.

fgets(): char \*fgets(char \*restrict s, int n, FILE \*restrict stream)

fgets() is the optimal solution for reading a large string of text from some input stream. Although fgets() was intended for use on files, it is the optimal solution for user input, as scanf() does not account for buffer overflow, and does not flush the buffer. Fgets() will only read a maximum of n-1 bytes, making it a safer option. An important thing to note is that fgets() will return less than n-1 bytes if it encounters a newline. Newline feeds are added to the output buffer. The reasoning behind this is pretty straightforward: fgets() stands for “file get string” i.e. it’s meant to return a string. Whatever bytes are read by fgets() are placed into the output buffer s.   
  
fputs: int fputs(const char \*restrict s, FILE \*restrict stream)

fputs() is equivalent to puts(), except that it can write to any file stream. It will write everything from s up to, but not including, the NULL termination character into the specified file stream.

fread: size\_t fread(void \*restrict ptr, size\_t size, size\_t nitems, FILE \*restrict stream)

fread is very similar to fgets(), but has a few key differences which might make it more useful than fgets() depending on the situation. To start, fread() returns size\_t. This represents the number of items that fread() successfully read. The second difference is that fread() takes a void\* as its first parameter, as opposed to char\*. This means that it is better suited for data buffers that aren’t necesarilly meant to be interpreted as characters (although, it obviously works for character buffers as well). A fourth parameter is added: nitems. nitems specifies the number of items to read. nitems is multiplied with size to get the resultant number of bytes that will be read by fread(). This is most useful when we are reading and writing data from a struct to a file. This will make more sense when we cover fseek() and ftell(). Essentially though, we can read n “nitems” of size “size” from the stream into our void\* buffer.

fwrite: size\_t fwrite(const void \*restrict ptr, size\_t size, size\_t nitems, FILE \*restrict stream);  
 As for fwrite(), this is the “write” counterpart to fread. It is useful for writing chunks of data at a time, such as a struct or array. We write nitems items of size size into the stream from our void\* buffer. fwrite() returns the number of items that it successfully wrote to the stream.

FILE and I/O Streams in More Depth:

Now that we’re familiar with some of the functions used for I/O, we can begin to understand how to operate on files. In Unix/Linux, we have devices (usually located in /dev/). These devices can be physical devices (graphics card, network card, RAM, disk, or other peripherals), or they can be virtual/pseudo devices (eg. /dev/random, /dev/null). These devices also have types (e.g. a mouse would be considered as a *character* device because it writes serial data one bit at a time, or a *block* device such as a hard drive which reads/writes data in sectors/pages). As I mentioned previously, file descriptors are integers which represent a file handle. The three file streams, stdin, stdout, and stderr appear as pseudo devices under /dev. For example, running ls -la /dev/stdout reveals that it is actually a link to /proc/self/fd/1. If we do an ls to list the other files within /proc/self/fd (ls /proc/self/fd) then you should see at least 3 symbolic links named: 0, 1, and 2. Naturally, these correspond to your three file streams. These are typically symbolic links to the same pts device (i.e. your terminal). For example, doing an ls -la /proc/self/fd/1 reveals that it is a symbolic link to /dev/pts/[n]. A pts is a pseudo terminal slave. A pseudo terminal slave is a slave to a pty. A pty is a terminal device which is emulated by another program such as xterm or ssh for example. On certain systems I pressume that you might find it pointing to /dev/tty rather than /dev/pts. All this to suggest that writing to the file descriptor 1 (stdout) would, in actuality, open the file /dev/pts/[n] and then read or write to it depending on the function call.

FILE struct:

The FILE struct is defined in stdio.h. It usually contains the following members (depends on implementation):

char \*\_IO\_read\_ptr; /\* Current read pointer \*/

char \*\_IO\_read\_end; /\* End of get area \*/

char \*\_IO\_read\_base; /\* Start of putback + get area \*/

char \*\_IO\_write\_base; /\* Start of put area \*/

char \*\_IO\_write\_ptr; /\* Current put pointer \*/

char \*\_IO\_write\_end; /\* End of put area \*/

char \*\_IO\_buf\_base; /\* Start of reserve area \*/

char \*\_IO\_buf\_end; /\* End of reserve area \*/

/\* The following fields are used to support backing up and undo. \*/

char \*\_IO\_save\_base; /\* Pointer to start of non-current get area. \*/

char \*\_IO\_backup\_base; /\* Pointer to first valid character of backup area \*/

char \*\_IO\_save\_end; /\* Pointer to end of non-current get area. \*/

fopen() and fclose():

In order to open and close files, we use the functions fopen() and fclose() respectively. The fopen() function takes in two parameters: A string containing the name of a file (this can be a relative path to the file, or absolute path from root), and a string representing the mode of operation. fopen() will return a file descriptor to the file. fclose() just takes in the file descriptor that we create using fopen().

Here are the various modes of operation that can be given to fopen():

**r:** Open a file for reading. If the file does not exist, fopen() returns NULL.

**rb:** Open for reading in binary mode. If the file does not exist, fopen() returns NULL.

**w:** Open file for writing. If the file exists, contents are overwritten. If it does not exist, the file is created.

**wb:** Open for writing in binary mode. If the file exists, contents are overwritten. If it does not exist, the file is created.

**a:** Open a file in append mode. Data is added to the end of the file. If the file does not exist, it will be created.

**ab:** Open a file for appending in binary mode. Data is added to the end of the file. If the file oes not exist, it will be created.

**r+:** Open for both reading and writing. If the file does not exist, fopen() returns NULL.

**rb+:** Open for both reading and writing in binary mode. If the file does not exist, fopen() returns NULL.

**w+:** Open for both reading and writing. If the file exists, its contents are overwritten. If the file does not exist, it will be created.

**wb+:** Open for both reading and writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.

**a+:** Open for both reading and appending. If the file does not exist, it will be created.

**ab+:** Open for both reading and appending in binary mode. If the file does not exist, it will be created.

Example of opening a file for writing:

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <stdlib.h>

#define DATA\_SIZE 50

int main(int argc, char \*argv[]) {

const char \*f\_name = NULL;

char data[DATA\_SIZE];

FILE \*fd;

char c;

unsigned i;

if (argc <= 1)

{

fprintf(stderr, "File: %s, Function: %s, Line %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

printf("Usage: %s <filename>\n", argv[0]);

printf("You must specify a file name as an argument when running this program\n");

exit(EXIT\_FAILURE);

}

f\_name = argv[1];

fd = fopen(f\_name, "w");

if (fd == NULL)

{

perror("Failed to open file: %s\n");

exit(EXIT\_FAILURE);

}

printf("Writing data into file %s...\n", argv[1]);

strcpy(data, "Wrote some test data into a file!!!\n");

for (i = 0; i < strlen(data); i++)

{

if ((c = fputc(data[i], fd)) == EOF)

{

perror("Error writing character from buffer to file\n");

exit(EXIT\_FAILURE);

}

}

printf("Finished writing!\n");

fclose(fd);

return EXIT\_SUCCESS;

}

fseek(), ftell(), rewind(), feof():

fseek() and ftell() are very useful functions when it comes to reading and writing to files. It may or may not have crossed your mind that a function such as getc()/fgetc() magically know what the next letter to read is. How is this accomplished? Well, we will learn about processes a bit later, but essentially, the process (our program) has a unique handle on the file that we call fopen() on. Recall that our FILE struct contains information about the current read/write pointers. These pointers point to location that we last read from, and the location that we last wrote to, respectively. These get incremented using pointer arithmetic everytime that they are accessed.

The functions fseek(), ftell(), and rewind(), all deal with moving the read/write pointer so that we can read/write to different locations in the file. Let’s begin with fseek().

fseek():int fseek(FILE \*stream, long offset, int whence)

fseek() is used to move the read/write pointer from some location “called whence”, to some offset location (whence + offset). It does this within the file specified by stream. Whence is measured in number of bytes from the start of the file (offset 0). Often times, we give the value SEEK\_CUR (defined in stdio.h) which is a macro that will tell fseek() to move the read/write pointer to its current position. For example, if we had already read 5 bytes, and we wanted to jump another 10 bytes ahead, we could either call fseek() like so: fseek(fp, 10, 5); or like so: fseek(fp, 10, SEEK\_CUR); Both are equivallent. Aside from SEEK\_CUR, there exist two other macros that are useful – SEEK\_SET and SEEK\_END. SEEK\_SET actually marks the beginning of the file (offset 0), and SEEK\_END marks the end of the file. Note that the value of offset can be negative, meaning that we can traverse backwards from whence. If we wanted to read the 1024th last byte of the file, we could say fseek(fp, -1024, SEEK\_END); which would set the read/write pointer of fp to the end of the file, and then backtrack 1024 bytes. If an error occurs, fseek() returns -1 and sets errno to one of EINVAL if the whence argument was not SEEK\_SET, SEEK\_END, or SEEK\_CUR, or ESPIPE if the file descriptor underlying stream is not seekable e.g. pipe, FIFO, or socket files.

ftell:long ftell(FILE \*stream)

ftell() is a much less complicated beast than fseek(). All it does is return the current read/write position of the file provided to it. This is useful if we need to poll our current position in the file for whatever reason. ftell() has the same error attributes as fseek().  
  
rewind:void rewind(FILE \*stream)

rewind(fp) is equivallent to fseek(fp, 0L, SEEK\_SET). All rewind does is set the read/write pointer back to the start of the file (offset 0). Rewind() has the same error attributes as fseek().

Let’s use our current knowledge to write a small program which will tell us the size of a file:

#include <stdio.h>

#include <stdlib.h>

/\* Function prototypes \*/

long get\_file\_size(FILE \*fp);

long get\_file\_size(FILE \*fp)

{

long int fsize;

fseek(fp, 0L, SEEK\_END); /\* Move r/w pointer to EOF \*/

fsize = ftell(fp); /\* Return current offset in bytes \*/

rewind(fp); /\* Move the r/w pointer back to the start of the file \*/

return fsize;

}

int main(int argc, char \*argv[])

{

FILE \*fp = NULL;

long fsize;

if (argc <= 1)

{

fprintf(stderr, "Error - No argument provided\n");

printf("Usage: %s <file\_name>\n", argv[0]);

exit(EXIT\_FAILURE);

}

fp = fopen(argv[1], "r");

if (fp == NULL)

{

perror("Unable to open file");

fclose(fp);

exit(EXIT\_FAILURE);

}

fsize = get\_file\_size(fp);

printf("The size of the file in bytes is %ld\n", fsize);

fclose(fp);

return EXIT\_SUCCESS;

}

feof():int feof(FILE \*stream);

feof() is essentially the same thing as doing if (getc(fp) == EOF). It returns a non-zero value if the current read/write position is set to EOF. feof() is sort of a convenience function to make things more readable, but checking if the current character equals EOF is equally valid. You could consider this a handy function for error checking when reading byte by byte.

opendir() and readdir(): DIR \*opendir(const char \*dirname);

struct dirent \*readdir(DIR \*dirp);

The opendir() function returns a file stream represented with the macro DIR (similar to the FILE struct but for directories). Keep in mind that directories on Linux are also considered to be files. A directory simply acts as a link to other file locations. Once the directory has beeen opened, we can read files from it using readdir(). This function takes in the DIR pointer that we give it, and returns a struct called dirent (directory entry) that contains a pointer to the next file in the directory. There is no guaranteed order when reading entries from a directory, so ordering must be done manually.

/\* dirent (directory entry) struct in Linux \*/

struct dirent {

ino\_t d\_ino; /\* inode number \*/

off\_t d\_off; /\* offset to the next dirent \*/

unsigned short d\_reclen; /\* length of this record \*/

unsigned char d\_type; /\* type of file; not supported by all file systems\*/

char d\_name[256]; /\* filename \*/

};

Randomness:

Often times, we need to have some sort of randomness within our application. As you may know, true randomness does not exist in software. Due to chaos theory, true randomness can be generated in hardware. Sometimes this is called TRNG (True Random Number Generation). TRNG modules can be costly, however, and since a reasonably good amount of pseudo-randomness can be achieved through software, most machines do not incorporate TRNG.

Randomness in software is achieved by scrambling an initial value known as the seed value using some sort of chaos algorithm. The output of the algorithm is usually fed back into the next invocation of the algorithm as the new seed value. I like to use the example of Minecraft seeds. When you go to create a world in Minecraft, you have the option of entering a seed value. If you enter the same seed value as another world, you end up with an exact copy of the other world. The reason that this is not true randomness is that inputting the same sequence of seed values will always produce the same results. There are various methods that people use to mitigate the odds of rolling the same sequence of random numbers. A common way is to use the system clock time as the initial seed value. This would be a good solution in most cases, however, we don’t always have access to a system clock. Another method is to invoke the random number algorithm upon triggering a specific event.

rand() and srand(): int rand(void);

void srand(unsigned seed);

We can return the next random number in the random number sequence by calling rand(). rand() returns a random number between 0 and RAND\_MAX (inclusive) which is at least 32,767. There is no way to specify a lower ceiling value within the parameter list, so you will often see people use the modulo operator to cap the return value. For example, if I only ever want random numbers between 1 and 10, I can say r = (rand() % 10) + 1. We require the +1 because a) rand() includes 0 by default, and we don’t want 0s, and b) the modulo excludes the 10 meaning that if we didn’t add the 1 as our floor value, we’d get numbers between 0-9.

Now, in order to set the seed, we use srand(). srand() takes in an unsigned int for the seed parameter. As I mentioned, we often pass in the current time since this will always be different everytime that you run the program. In order to get the time, we can #include <time.h>, create a time\_t typedef, and then call time(time\_t \*t). This should get cast to an unsigned int to match the parameter of srand(). Here’s an example:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <unistd.h>

#define RFLOOR 1

#define RCEIL 10

int main(void) {

int r;

time\_t t;

srand((unsigned) time(&t));

for(;;) { /\* Loop forever \*/

r = (rand() % RCEIL) + RFLOOR;

printf(“Random number is: %d\n”, r);

sleep(1);

}

return 0;

}

The sleep() function just makes the program wait 1 second between each loop (we’ll cover sleep() later in the document). In case you haven’t seen for(;;) before, it’s essentially equivallent to while(1). Basically, a blank for statement evaluates to true, and is therefore another way of writing an infinite loop. We start the program by creating a time\_t typedef (pressumably an int, but depends on the system) which gets set by calling time() with a pass by reference. This value will set the seed to something different everytime we run our program. rand() then returns a number between 1 and 10 in the infinite loop and prints its value every second.

Kernel/System Calls:

Hopefully you are aware of what system calls are, but if not, no worries; I’ll give a quick breifing. Kernel/System calls are really just functions that are normally wrapped by higher level functions in libc. We are able to invoke kernel/system calls from user mode, but the key point is that they are executed in kernel mode (i.e. a higher privilege). System calls are typically related to system functionality (things like opening files, reading and writing, forking processes, etc.). Check the syscalls man page for a list of system calls.   
  
open: int open(const char \*pathname, int flags)

The open() kernel/system call is used to open a file. But doesn’t fopen() do that? Well yes, but fopen() builds upon open(). The most important distinction between open() and fopen() is that fopen() returns a file stream, whereas open() returns a file *descriptor*. The difference between these can often be confusing and complicated. I will try to be abundantly clear here because it is quite easy to find misleading information on the internet. A file stream is a conceptual idea based around the FILE struct and the f\*() API functions such as fopen(), fwrite(), fgets(), fread(), etc. The FILE struct is a wrapper for a file descriptor that contains additional information than that of a file descriptor. A file descriptor appears to you and I to be an integer value. Whenever you receive a file descriptor, you treat it as an int in C. That is only half the story!!! Although to you, the end user, a file descriptor may *seem* like just an integer, this is only half true. File descriptors are really IDs or handles to what is formally called an Open Control Block (OCB) on Unix/Unix-like systems. An OCB is maintained by the filesystem manager and contains per-open data. That is to say, it contains similar data to that of the FILE struct. Things like the latest access date, the file read position pointer, the file’s inode, etc. Essentially this contains all information about the file. In C, we do not see any of this. All we see is that file streams are defined as FILE structs, and file descriptors are returned to us as integers. A good illustration of this is that in C, we have stdin, stdout, and stderr, which are our standard I/O streams (defined as FILE\* structs), but we also have the macros STDIN\_FILENO, STDOUT\_FILENO, and STDERR\_FILENO which are defined as 0, 1, and 2 respectively. Note that we can obtain the file descriptor of a file stream using the function fileno(). For example, fileno(stdout) returns 1. One final note that I think is quite important: Processes share file descriptors. What I mean by that is that once a file descriptor is registered, the OCB has already been created. If we open the same file descriptor on another process, we open the same file. The system registers file descriptors 0, 1, and 2 at boot, meaning that they are permanently reserved, and we cannot use them. Luckily, file descriptors are managed by your system, so a call to open() returns the lowest available file descriptor. We can duplicate a file descriptor using a function called dup(). What this does is generate a new **handle** i.e. integer value to point to an existing OCB, but it does not duplicate the OCB itself. As an example, file descriptors 5 and 7 may point to the same OCB if dup() creates fd 7 as a duplicate of fd 5.

Because open() only returns a file descriptor, we cannot use any of the file I/O functions that we’ve covered thus far, unless we convert the file descriptor to a FILE pointer using fdopen(). Another unique thing about open() compared to fopen() is that open() gives us a large range of flags to choose from. These can be OR’d together to apply multiple attributes to the file descriptor. They include, but are not limited, to the following:

- O\_RDONLY: Open the file to be read only

- O\_WRONLY: Open the file to be write only

- O\_RDWR: Open the file for read and write operations

- O\_APPEND: File is opened in append mode

- O\_ASYNC: Enable signal-driver I/O. May lead to corrupt files on NFS systems

- O\_CLOEXEC: If set, the file descriptor is closed whenever one of the exec\*() family of functions is called. The exec\*() functions are covered in my POSIX programming notes.

- O\_CREAT: If the pathname does not exist, create the file specified in pathname

- O\_DIRECT: Minimize cache effects of the I/O to and from the file

- O\_DIRECTORY: If pathname is not a directory, cause the open to fail

- O\_EXCL: Same as O\_CREAT. If ORd with O\_CREAT, returns EEXIST if the file already exists, rather than overwriting it

- O\_LARGEFILE: Allow files whos sizes cannot be represented in an off\_t typdef, but can be represented as an off64\_t

- O\_NOATIME: Do not update the access time when the file is read

- O\_NOCTTY: If the pathname refers to a tty, it will not become the processes controlling terminal, even if the process does not have one

- O\_NOFOLLOW: If pathname is a symlink, the open fails with the error code ELOOP

- O\_NONBLOCK/O\_NODELAY: The open() function, nor any further I/O calls will cause the process to wait

- O\_PATH: Treat the file as a location in the filesystem tree. The file does not actually get opened if O\_PATH is specified, and trying to open it will return EBADF

- O\_SYNC: Will cause all I/O operations to be synchronized

- O\_TMPFILE: Create an unnamed temporary file

- O\_TRUNC: If the file already exists and has r/w permissions, this will truncate the file to length 0

close: int close(int filedes);

Have you ever wondered why closing files is actually necessary? Well the primary reason is because close() flushes the write buffer so that any data lingering in it gets written to the currently opened file. I was recently creating an application for a customer that would listen for keyboard interrupts and then convert the key/scan codes into ASCII characters and write those characters to a file to be read. This was happening in a while loop and anytime that an interrupt came in, the key code was processed and written to the file. Another client application was supposed to be able to read from the file at any time, but when I would attempt to do a read, no data would come back from the file. This was because I was not flushing the write buffer so nothing was actually getting written to the file. Before I knew about fsync() the solution was to call open() at the top of the while loop and close() at the bottom. This works as expected (albeit quite inefficient). fsync(), in case you were wondering, is essentially the equivallent of fflush() but for file descriptors.

read and write:

ssize\_t read(int filedes, void \*buf, size\_t nbyte);

ssize\_t write(int filedes, const void \*buf, size\_t nbyte);

The read() and write() system calls are useful for, well, reading and writing. More specifically though – read() and write() are used to read/write to a file given a file descriptor. Both sys calls take in a buffer as their second parameter and the number of bytes to read/write as their final parameter. These sys calls are sort of annoying to deal with because they can be interrupted by signals or other things. They both return the number of bytes read or written respectively. EOF is equivallent to 0 because POSIX states that when there are no more bytes left to read in a file, read() must return 0 bytes read (EOF). Here is an example of using open(), followed by write() and read():

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

#define OUT\_MAX 256

#define IN\_MAX 256

/\* Function declarations \*/

char \*ret\_usr\_input(char \*input);

char \*ret\_usr\_input(char \*input)

{

printf("Please enter some data to write to the file: ");

input = fgets(input, IN\_MAX, stdin);

fflush(stdin); /\* Flush anything left in the buffer \*/

return input;

}

int main(void)

{

int fd;

ssize\_t nbytes\_r, nbytes\_w;

void \*bufin = NULL;

const void \*bufout = NULL;

char input[IN\_MAX];

/\* Ask for user input \*/

bufout = (void \*)ret\_usr\_input(input);

if (strcmp(bufout, "\n") == 0)

printf("Warning: Nothing was entered...\n");

else

printf("User entered: %s\n", bufout);

/\* Open file and set the file descriptor \*/

fd = open("tempfile", O\_RDWR | O\_CREAT | O\_TRUNC, 666);

if (-1 == fd)

{

fprintf(stderr, "Open failed: %s", strerror(errno));

exit(EXIT\_FAILURE);

}

/\* Write some data to the file \*/

while ((nbytes\_w = write(fd, bufout, strlen(bufout))) != strlen(bufout))

{

fprintf(stderr, "Failure - Partial write: %s\n", strerror(errno));

}

sleep(1);

printf("Data was successfully written to file!\n");

printf("Contents of file are:\n");

/\* Make sure to reset file pointer for read op \*/

lseek(fd, 0, SEEK\_SET);

bufin = malloc(sizeof(char) \* IN\_MAX);

if (bufin == NULL)

{

/\* Read data from file and send it to stdout \*/

while ((nbytes\_r = read(fd, bufin, IN\_MAX)) > 0)

{

if (write(STDOUT\_FILENO, bufin, nbytes\_r) != nbytes\_r)

{

fprintf(stderr, "Failure - Partial write: %s\n", strerror(errno));

fprintf(stderr, "File: %s, Function: %s, Line: %d\n", \_\_FILE\_\_, \_\_func\_\_, \_\_LINE\_\_);

exit(EXIT\_FAILURE);

}

}

if (-1 == nbytes\_r)

{

fprintf(stderr, "Failed to read data from file: %s\n", strerror(errno));

exit(EXIT\_FAILURE);

}

if (-1 == close(fd))

{

perror("close()");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

Now, this looks very scary, but I promise it’s not that bad. I recommend that you copy and paste this into vim or an IDE of your choosing because you will be able to see it all at once much easier. I won’t go into detail about all the pointer stuff – you should hopefully be able to follow along with that. The main parts that I want to touch on are the calls to open(), read(), and write(). First, with open(), we create a file called tempfile and return a file descriptor. The option flags are set such that a new file will be created if it does not exist (O\_CREAT) and upon creation, it will have read/write permissions (O\_RDWR). The O\_TRUNC option makes it so that on open, the file is cleared, which is what I happen to want for this program. I also pass it the value 666, which is not the mark of the beast, but rather the file permissions rw-rw-rw on Linux. This value may change when you run this due to your umask, so you may need to manually set the permissions with chmod. Anyways, for writing to the file, we call write(), but we put it in a while loop. This is so that the function will re-run if it only writes partial data, and not the whole contents of bufout. This same methodology applies to read() as well. We continuously call read() while it returns > 0. This is again, because read() returns the number of bytes read and we want it to stop when it reads no bytes (0 is returned if read() encounters EOF).

fstat(): int fstat(int fildes, struct stat \*buf);

fstat is likely the simplest way of obtaining information regarding a file given a file descriptor. The file’s information will be stored in a stat struct referenced by buf. E.g:

int fdopen();

struct stat file\_stat;  
fstat(

mmap, munmap, mremap:

void \*mmap(void \*addr, size\_t len, int prot, int flags, int fildes, off\_t off)

int munmap(void \*addr, size\_t len)

void \*mremap(void \*old\_address, size\_t old\_size, size\_t new\_size, int flags, ...)

malloc() is the standard allocator provided by libc. Although it is the most commonly used method for allocating additional memory, it is not the only way. The mmap(), munmap, and mremap() system calls are alternatives for malloc(), free(), and realloc(), respectively. The basic idea behind mmap is to map physical addresses to virtual addresses. mmap() supports regular files, shared memory objects (shared memory), and typed memory objects (a.k.a. POSIX typed memory). Support for other file types is unspecified. I’d say that there are three primary ways that we can use mmap(). The first is to create a direct mapping to some physical address i.e. map directly from PAS to VAS by providing mmap() with an explicit physical address. It is extremely discouraged to create a direct physical mapping that is shared between processes. You would typically use a direct mapping if you plan on interfacing with hardware registers of an I/O device, for example. Note that a physical mapping is by definition also what we call an anonymous mapping. That is to say, it has no association with any file. The second method of using mmap() is to give no file descriptor, or physical address, which lets the kernel allocate some physical space for us whenerever it chooses on the system and then creates the virtual mapping. The final common way of using mmap() is to use an existing file as the memory backing. We can provide an explicit file by passing a file descriptor to mmap() and mmap() will create a virtual mapping using the file’s physical entry address (starting location) and its size. An important note here is that files are stored on hard disk, not RAM, meaning that mmap-ing memory using a file descriptor is equivallent to creating swap memory (memory on disk). Using the latter two methods, we can create a shared mapping, which simply means that the MMU will allow other processes to touch the memory so long as they abide by the R/W/E permissions that we set.

The addr parameter specifies the static address of the new mapping. If addr is NULL, then mmap() will still try to create a new mapping, but the address will depend on where the kernel decides to place it (this can pretty much be anywhere in RAM, not including device memory i.e., memory addresses that actually point to hardware). The kernel will ensure that it is page-aligned, and therefore, setting addr to NULL is the most portable way of using mmap(). The static address must be a multiple of the system page-size which can be returned using the sysconf() system call with the argument \_SC\_PAGE\_SIZE eg. sysconf(\_SC\_PAGE\_SIZE). Page size on modern 64-bit processors is usually 4096 bytes. We can check that the return addrss of mmap() is page aligned like so: *assert(addr % sysconf(\_SC\_PAGE\_SIZE) == 0)*.

The len parameter specifies the size of the mapping in bytes. This length will be relative to the offset parameter, off.

The prot parameter of mmap() accepts a combination of enums (using logical OR) to specify the mode of operation of the memory that is being mapped. These enum values are as follows:

- **PROT\_EXEC:** Pages may be executed

- **PROT\_READ:** Pages may be read

- **PROT\_WRITE:** Pages may be written

- **PROT\_NONE:** Pages may not be accessed

The flags parameter determines how other processes may communicate with the memory. POSIX defines at minimum the following enums (although there may be many more depending on the implementation of libc on your system):

- **MAP\_SHARED:** Create shared memory. Other processes are able to access the memory, but still have to comply with the protection rules specified in the prot parameter.

- **MAP\_PRIVATE:** Create a private Copy on Write (COW) mapping. Multiple processes are able to access and modify this memory, but the changes are not synchronized to disk, meaning that it appears as though only the parent process is able to access/modify this memory.

- **MAP\_FIXED:** MAP\_FIXED forces the mapping to be placed at the address specified by addr. It is implementation defined whether or not MAP\_FIXED is supported. POSIX defines it, but that doesn’t necessarily mean that it is enforced on your system. This flag must be used if the addr parameter is not set to NULL.

Passing in a file descriptor for the filedes parameter will cause mmap() to use that file as the memory backing. The easiest way to retrieve a file’s file descriptor is by using the open() system call. Setting the fildes parameter to -1 will create an anonymous mapping.

The offset parameter, called off, specifies the starting location of the mapping relative to addr. The offset must be a multiple of the system page size.

The munmap() function is what free() calls to mark the memory as available. This will allow it to be reused by the parent process. Similar to mmap(), it takes an address and length. These should match those of the mapping that you are unmapping.

Let’s look at an example of how we could write a simple malloc() function using mmap():

void \*my\_malloc(size\_t size) {

static int fd\_zero = -1;

if (-1 == fd\_zero) /\* Only open() on first call to malloc() \*/

fd\_zero = open(“/dev/zero”, O\_RDWR);

void \*p = mmap(NULL, size, (PROT\_READ | PROT\_WRITE),

MAP\_PRIVATE, fd\_zero, 0);

if (MAP\_FAILED == p) { p = NULL; }

return p;

}

As you can see, this is very concise for such a useful function. First, we create a static file descriptor called fd\_zero. File descriptors are always > 0, so -1 falls outside the range of valid file descriptors. Note that fd\_zero is static, meaning that it will exist for the duration of the program. The first time that this version of malloc() is called, fd\_zero is declared and initialized to -1, but on consecutive runs, malloc() will not redeclare it, and thus, will not re-initialize it. But let’s say that this is the first time calling malloc() in our program. fd\_zero is initialized to -1, and is set to the return value of open() (which returns a file descriptor associated with the file /dev/zero). We specify that this file descriptor is O\_RDWR, meaning that we can read and write to/from it. Next, we call mmap() to map some memory. We give NULL for the address so that the kernel will do that for us, and ensure that it is placed in accordance with the page-size. We specify that it will be PROT\_READ and PROT\_WRITE, i.e., no external processes should be able to read or write to it. MAP\_PRIVATE basically prevents external processes from even trying to access it at all, and tells mmap() to make the memory copy-on-write. We give fs\_zero as the argument to fildes so that we use it as the physical memory backing for the virtual memory address that will be returned. Finally, we pass 0 in for the offset since we don’t require an offset. A basic if statement checks for a failed mapping with MAP\_FAILED, and sets the pointer = NULL if it failed. It then returns the address pointed to by p. Note that MAP\_FAILED was defined in POSIX.1-2017, and that earlier implementations of mmap() return (void \*)-1. If you are not aware, /dev/zero on Unix-like systems is a special character device file which is mapped to some hardware that returns a stream of 0s. For this reason, this implementation of malloc() acts more like calloc() than libc’s implementation of malloc(). We can see this by printing out each byte of the allocated memory:

int \*p = my\_malloc(5 \* sizeof(int));

for (int i = 0; i < 5; i++) {

printf("0x%X ", \*(p+i));

}

printf("\n");

Outputs: 0x0 0x0 0x0 0x0 0x0

Finally, let’s look at mremap(). This function is what gets used by realloc() to reallocate memory. In reality, we are actually just mapping in more physical address pages to virtual ones, or unmapping existing pages depending on whether or not the new size is greater or less than the original size. The flags parameter may either be 0 or one of the following:

- **MREMAP\_MAYMOVE:** Allows the kernel to move the base address of the mapping if the kernel does not have a sufficient amount of space to expand the memory object at its current mapping.

- **MREMAP\_FIXED:** Similar to MAP\_FIXED flag of mmap(), MREMAP\_FIXED allows for the use of an additional fifth argument (void \*new\_address), which specifies a page-aligned address to which the mapping should be moved. In other words, it becomes the new base address of the memory object.

- **MREMAP\_DONTUNMAP:** Used in conjunction with MREMAP\_MAYMOVE, remaps a mapping to a new base address but does not unmap the old address. This can only be used with private anonymous mappings

brk and sbkr:

int brk(void \*addr)

void \*sbrk(intptr\_t increment);

Similar to mmap(), brk (pronounced break) and sbrk (set break) change the location of the program break (the process’ .data segment). By moving the location of the break up or down in the stack, we effectively increase or decrease our program’s memory. Increasing the location of the break has the effect of allocating memory, and decreasing it has the effect of deallocating memory. brk() is the function which sets the program break to the address specified in the addr parameter. sbrk() on the other hand, increments the program’s data space by the number of bytes specified in the increment parameter. This can be a negative value which will subtract from the current value. Since sbrk() returns the new address of the break on success, giving it 0 as the argument for increment has the effect of returning the current break address e.g. void \*curr\_brk = sbrk(0); brk() returns 0 on success. Both functions return -1 on failure and set errno to ENOMEM.

For a more detailed look into malloc, mmap, brk, and the differences between them, I highly recommend the youtube video “The Origins of Process Memory | Exploring the Use of Various Memory Allocations in Linux C” by Low Level Learning.

Compiling From Command Line:

At this point in the document, you’ve reached the end... Of the C programming part (although check my POSIX programming notes to learn more about system programming using the POSIX API)... We still need to discuss compilation from the command line, which is a massive topic to cover on its own, unfortunately. In fact, I would argue that mastering the command line tools for compiling programs in C is almost half of the skill it takes to be an experienced C programmer. “Isn’t there a GUI I can use or something to make compiling easy?” you may be asking. Well, sort of. IDEs can often do a lot of the dirty work of compiling for you, but unless you understand how that process works behind the curtains, you will be worse off for it. Don’t fret, I hope to help make the process as painless as possible.

Compiling a Single File Application:

Let’s start off basic, and assume that you only have one .c file. Let’s call it main.c. In Linux, you have access to the GNU C Compiler (gcc) toolchain. GCC will be the utility that we use to compile our source files into executable binaries. On Windows, you do not have access to gcc, so you must install either Cygwin or MinGW which are Linux terminal emulators that can run natively on Windows, and then install gcc. With the rise of Windows 11, I’ve heard that Windows Subsystem for Linux (WSL) is also improving rapidly (that’s what I’ve *heard*), which might also be a viable solution for you. Or just install Virtual Box, VMWare, QEMU, or whatever Virtual Machine management tool you prefer and throw a Linux ISO on there. Once you have access to gcc, we can simply run the command “gcc main.c” which will compile our source code and produce a file called a.out. The history of a.out is a story that you can look up yourself, but essentially it’s an old file format that is no longer used, the name just stuck around. You can run that executable as you would any other executable in Linux e.g. “./a.out”.

Changing the Binary Name:

GCC has many compiler options, a common one being -o which stands for “output name”. By specifying the -o option with a value, we can change the name of the executable from a.out to whatever we want. For example, gcc main.c -o helloworld for a hello world program.

Compiling Multiple Files:

Let’s say now that you have a .c file and a .h file, or perhaps multiple .c and .h files. That is easy enough, we just add each file that will be linked during the linking phase in our gcc command. For example, let’s say that we have a source and header called foo.c and foo.h respectively, as well as a source and header called bar.c and bar.h. In order to compile all of these into one binary, we run the command: gcc foo.c bar.c foo.h bar.h -o baz. In this example, the ordering of the files does not matter, nor does the placement of the -o option. This example would produce an executable called baz.

Compiling With Debug Symbols:

In the final section of this document, we’ll cover debugging with GDB, which is the GNU Debugger. You will likely hear the term “debug symbols” at some point in your programming career. This refers to the symbols that are stored in executables that provide the user with extra information. We lightly touched on this before, but running objdump on a .elf executable will dump the contents of said executable. Most notably, this contains variable names and function names. These can be “stripped” i.e., removed from an executable by using the atply named “strip” command. Whenever a product is shipped, it is generally a good assumption that debug symbols will be stripped. Not only does this significantly reduce the size of executables, it also acts as a safety measure for proprietary code, making it harder to reverse engineer (not that you should be using proprietary software of course ;) The downside to stripping debug symbols is, well... it pretty much becomes impossible to debug our code! By default, gcc does not include debug symbols, so it is always a good idea to add them, especially during a pre-release build. Long story short, we add debug symbols with the -g option. This simple option will turn on debug symbols, so you might as well always add them, especially since they can be stripped later if necessary.   
  
Compiler Optimization:

One duty of the compiler is to optimize the low-level code if the user requests that it do so. There are various levels of optimization which will increase performance at the risk of introducing bugs. The levels are as follows (according to the official GNU docs):

-O0: No optimization (the default); generates unoptimized code but has the fastest compilation time. Note that many other compilers do substantial optimization even if ‘no optimization’ is specified. With gcc, it is very unusual to use -O0 for production if execution time is of any concern, since -O0 means (almost) no optimization. This difference between gcc and other compilers should be kept in mind when doing performance comparisons.

-O1: Moderate optimization; optimizes reasonably well but does not degrade compilation time significantly.

-O2: Full optimization; generates highly optimized code and has the slowest compilation time. O2 is considered to be the last “safe” optimization level.

-O3: Full optimization, as in -O2, also uses more aggressive automatic inlining of subprograms within a unit ([Inlining of Subprograms](https://gcc.gnu.org/onlinedocs/gnat_ugn/Inlining-of-Subprograms.html" \l "g_t100)) and attempts to vectorize loops. O3 is prone to break code, and is usually not used in favor of O2.

-Os: Optimize space usage (code and data) of resulting program.

-Ofast (not in docs): -Ofast will attempt to optimize for speed. This is supposedly quicker than -O2 and -O3.

-Og (not in docs): The -Og flag attempts to optimize for debugging purposes, shortening compile times and increasing performance while debugging.

As a side note, I generally use -O0 for debug code, and -O2 for release code. The slowdown of compiling with -O2 or -O3 may be significant for much larger projects which you may want to take into consideration when building them.   
  
Warning Flags:

Not many new C programmers enjoy using the warning flags because they feel like nothing but endless nagging. However, these flags will save you a lot of trouble, so I highly recommend that you use them. There are a ton of warning flags, each of which start with -W, but we will go over just a few that I use consistently: -Werror, -Wextra, -Wall, -Wformat, -Wconversion, and -Wpedantic.

-Werror: This option is generally good to have on, though it can be a bit annoying in certain cases. The Werror flag tells the compiler to treat warnings as compile time errors which cannot be resolved unless the programmer explicitly fixes them. This can be annoying if you’re doing things like multithreading, because threads require you’re function signature to contain a parameter that often goes unused, and unused parameters are considered as warnings by GCC. With the Werror flag enabled, you either have to set a pragma directive to ignore those specific warnings, or use the parameter in some way.

-Wextra: This option pretty much does what it says – it enables some extra warnings aside from your typical error warnings. I highly recommend that you enable this flag if you are not using the next option in the list.

-Wall: -Wall turns on pretty much all of the warning options. You can view these by looking at the man pages for gcc. -Wall can be very annoying and nag about seemingly very silly things like unused code, but I guarantee that using this will prevent a lot of headaches. Personally, I use -Wall for my own projects whenever I can.

-Wformat: The Wformat flag checks calls to printf and scanf to make sure that the arguments supplied make sense in relation to their corresponding format specifiers. For instance, if you’re passing a long int into printf(), when the corresponding format specifier is %d (regular int), gcc will print a warning assuming that the Wformat flag is enabled.

-Wconversion: Wconversion checks for things such as integer overflows or type-cast truncation. Implicit casts to a narrower type e.g. and int being cast to char or overflows e.g., passing UINT\_MAX + 1 as an argument for an unsigned int parameter will generate a warning with this flag enabled.

-Wpedantic: This option goes even further than -Wall and forbids anything that is not part of the ISO C standard. I recommend using this as well, unless you really know what you are doing. This generally prevents you or others from doing weird stuff that is not defined in the C standard, making your code safer and more portable.

Changing the C Standard:

While we’re on the topic of C standards, we can change which one that we’re complying with. This is done with the -std=standard option, where standard is one of the following: c89, c90, c99, c11, c17, c18, gnu89, gnu90, gnu11, gnu17, gnu18, gnu2x, and then some other C++ options that we wont get into. The default is surprisingly gnu17. I’m not quite sure why this is, but it’s not that important. The main thing is that if you’re trying to write portable code, especially for older hardware, I recommend turning on -std=c89 or -std=c90. Some people use the -ansi flag, which is equivallent to -std=c90 for C programs and -std=c++89 for C++ programs, however, -ansi will also replace certain functions such as sinf, cosf, and tanf from libm, which can be extremely annoying when debugging, so I don’t recommend using it.

Outputting Intermediary Compile-Time Files:

We are able to output intermediary files during the various stages of the compilation pipeline. Specifically, we can gather the output of the preprocessor with the -E option, the output of the assembly code with the -S option, and the output of the compiled code (pre-linker) with the -c option. We can then output these to their own files. For example, “gcc -E main.c > main.i” outputs the preprocessor data into a file called main.i (the .i extension is convention). gcc -S main.c > main.s would output the assembly code to main.s (again, .s is a convention). And last but not least, gcc -c main.c would produce main.o. In order to do all three of these at once, we can use the -save-temps flag to produce a .o, .i, and .s file. For example, “gcc -save-temps main.c” would produce the following files: a.out, main.o, main.i, and main.s.   
  
Linking Libraries:

Now we’ll have to go a bit in depth into how libraries work... Most of the libraries that you see in Linux and Windows (and probably Mac as well) are Dynamic Link Libraries (DLLs). In Windows, DLLs have a .dll extension, but in Linux, DLLs go by an alternate name, shared objects, and have a .so extension (and on Mac, they are called .dylib, short for dynamic library). There do also exist static libraries which have a .a extension in Linux and a .lib extension in Windows. Library files always start with “lib”, followed by the name of the library, and then the extension. Sometimes the version of the library also comes after the extension. For example, libc.so.6, or libcrypto.so.1.1. This is so that you can have multiple versions of a library on your computer in case older software still uses the old library as a dependency. A common trick is to have a symlink file point to the correct version. For example, if you had libc.so.1, libc.so.2, and libc.so.3 on your computer, you might have a soft link called libc.so that points to libc.so.3, but can change depending on the environment. Always use ls -la to check if a library is a symlink to another file. Anyways, in order to link your program to these files, we use the -l flag, followed by the name of the library. Since all libraries have the “lib” prefix, we drop that when using the -l option. For example, you may need to link with libm.so if you are using functions from math.h, so we would compile our program like so: gcc main.c -o helloworld -lm. Note that we don’t have to link with libc becuse it is linked automatically for us. If we do not wish to link with libc, we can use the -nolibc flag. Also note that static libraries do not need to be linked with with the -l flag since they are included in the list of files amongst our source files and header files when compiling. You can, however, specify which directories to look in for library files with the -L flag. For example, if we wanted to compile our program with a static library called libstatic.a, and it was in project/lib/ (ie. not in our $PATH), then we could compile like so: gcc main.c libstatic.a -o helloworld -Lproject/lib. Even though libstatic.a is not in our present working directory, the compiler finds it in project/lib/ because we told it to search there. Note that gcc searches your $PATH environment variable for libraries by default, which is where most system libraries are stored.

Searching Directories for Header Files:

Similar to the -L flag, we can also use the -I flag (stands for include) to search a list of directories for header files. For example, it might be cleaner to store our header files in project/include/, so we can specify -Iproject/include so that the compiler searches that directory. I didn’t mention this for the -L option, but we can tell both options to search in multiple directories using the colon separator, similar to how you would separate directories in your PATH. For example: -Iproject/include:other/dir would search both project/include/ and other/dir/.

Verbose Compiling:

Many applications on Unix-like systems use the -v flag to specify verbosity. Sometimes adding more vs increases the levels of verbosity. In gcc, there are only two levels of verbosity: the default, and verbose, specified with -v. Doing so will output a lot of stuff that you may or may not understand, but a lot of it is quite useful. For instance, there is information about what libraries were linked, what directories it searched for them in, as well as what flags were used behind the scenes, etc.   
  
Defining Macros on the Command Line:

At first, this option may be confusing to you, or you may not understand what sort of purpose it might serve, but the -D flag can be very useful, as it is used to define macros during compile time. This can essentially act as a context switch. I’ve often seen -DDEBUG used, which #define(s) a macro called DEBUG. Somewhere in the code, we can check if DEBUG has been defined with the #ifdef directive and then execute some code accordingly. Eg.

#ifdef DEBUG

if (val1 > val2)

printf(“Value 1 is bigger than value 2\n”);

#endif

Creating Position Independent Code/Executables:

If we are creating a shared library, it is important that our code be position-independent. This means that it does not need to start at a specific address in RAM. Similarily, if we need to load an entire executable into RAM, we can do so. Both of these features have the added benefit of Address Space Layout Randomization (ASLR), which makes it more difficult for attackers to locate the starting position of your program or shared object and exploit it. We can use the -f option in gcc followed by PIC (position-independent code) or PIE (position-independent executable) depending on if we’re creating a shared object, or an executable that gets loaded into memory e.g., -fPIC or -fPIE.

Creating a Static Library:

Static libraries have rapidly been losing their appeal due to the unfortunate draw-back of making code too large. To illustrate this, imagine that libc was a static library. Let’s also say that libc was 500MB in size. This might be acceptable for one program, but since virtually all C programs link with libc, every process on your computer would be taking up an additional 500MB of space. If you had 500 processes running, then you’d have 250GB taken up by libc!!! None-the-less, there are valid reasons for using static libraries, especially for the ease-of-use, so it’s good to know how to build one. Creating static libraries is unique in that we don’t actually do it using gcc. Instead, we use a command line utility called ar which stands for archive. This makes more sense when you realize that static libraries are really just archives that contain a bunch of object files. Assuming that we have 3 object files, file1.o, file2.o and file3.o, we can create a static library as follows: ar rcs libstatic.a file1.o file2.o file3.o. The usage of the ar command is a bit strange but essentially, rcs is actually 3 separate subcommands. The r stands for replace ie. replace an existing library with the same name if it exists, the c option stands for create ie. create the static library if it doesn’t exist, and s means index (not sure why it’s an s and not i) i.e. create an index which tells the compiler that it’s a library file that we’re creating. Note that the library is the first file in the list of all files and then each proceeding file is assumed to be an object file that will be added to the archive. It is important that the name of our library starts with the prefix “lib” because another compiler flag, -l, assumes that all libraries start with that prefix. In order to link with our static library, we can either add it in the dependency list when compiling with gcc (along with the other object files), or we can link using the -l option in gcc eg. -lstatic means “link with libstatic.a”.

Creating a Dynamically Linked Library (DLL):

As we’ve covered already, DLLs (which are represented as shared object files in Linux) are libraries that are constantly stored in RAM for us to link with during the loader phase of the compilation pipeline. These have many benefits such as keeping our binaries small as well as the fact that they are more easily updated/maintained. In order to create a shared object file, our code must be position independent, since we want to give it flexibility as to where it’s stored in RAM (and because we don’t want to allocate space for it if it is not being used). This is accomplished using the -fPIC option in gcc. Another important flag is the -shared flag which specifies that we are creating a shared object. In order to create the shared object file given 3 source files – file1.c, file2.c, and file3.c, we can simply run gcc -fPIC -shared -o libshared.so file1.c file2.c file3.c. Unlike static libraries, we can’t just link a shared object file by adding it to the dependency list as we would with our object files. We *must* use the -l flag. In this case, we’d do -lshared to link with libshared.so.

Dynamic Linking at Runtime:

Typically, the loader will load both our program and unloaded libraries into memory and then jump to program execution. This is sufficient for most programs; however, it is possible to load and unload libraries during the runtime of our program. The benefits of this may not be immediately apparent, but I assure you that this can be ideal in many circumstances. By splitting our code into multiple DLLs, we can create extremely modular code that can be altered on the fly. An example of how this could be useful can be illustrated with videogame development. Let us say that you are testing a boss fight, adjusting the damage output of certain attacks and tweaking things as you play around. Every time that you want to change a value, you are forced to exit the game, change the value in the code, recompile, and then get back to that point in the boss fight. In order to solve this, the boss’ code can be stored in a DLL/shared object. By recompiling the DLL, the behavior of the boss can be changed on the fly, without having to halt the program. The functions that we will be looking at to accomplish this belong to dlfcn.h.  
  
dlopen: void \*dlopen(const char \*filename, int flag)

dlopen(), as you may be able to guess, is the function which loads the dynamic library file using the null-terminated string filename. The function will return a handle for the dynamic library upon finding the dynamic library. If filename is NULL, dlopen() returns a handle for the main program. If pathname contains one or more slashes then it will be interpreted as either a relative path or absolute path, otherwise the linker searches for the library using a specific algorithm which I wont touch on here.

One of the following two flags must be passed into the flag parameter. The two flags are RTLD\_LAZY and RTLD\_NOW. RTLD\_LAZY does lazy binding (similar to copy-on-write). Lazy binding only resolves symbols at the time that they are referenced. If they are never referenced, they are not resolved. With RTLD\_NOW, all undefined symbols must be resolved before dlopen() returns. If the linker fails to resolve all symbols, an error is returned.

Zero or more of the following flags can be ORd together with the flag chosen above. Here is the list of flags available:

**RTLD\_GLOBAL:** This flag makes it so that symbols defined in the library being loaded will be available for use in the next library that is loaded.

**RTLD\_LOCAL:** Contrary to RTLD\_GLOBAL, RTLD\_LOCAL marks symbols defined in the library being loaded as local only. RTLD\_LOCAL is the default option if neither RTLD\_GLOBAL or RTLD\_LOCAL are specified.

**RTLD\_NODELETE:** This flag makes it so that the library will not be unloaded during dlclose(). Static variables will not be reinitialized if the library is reloaded with subsequent dlopen()s.

**RTLD\_NOLOAD:** This flag makes it so that dlopen() wont actually load the library at all. This is used to test if the library has already been loaded, since dlopen() returns NULL if it is not, or the handle if it is.

**RTLD\_DEEPBIND:** This flag puts the lookup scope of symbols in the library ahead of the global scope. This means that a self-contained library will use its own symbols in preference to global symbols that share the same name.

dlerror: char \*dlerror(void)

dlerror() returns a human readable string describing the most recent error that occurred from dlopen(), dlsym(), or dlclose() since the last call to dlerror(). It returns NULL if no errors have occurred.

dlsym: void \*dlsym(void \*handle, const char \*symbol)

dlsym() takes a handle of a dynamic library loaded by dlopen(), as well as the null-terminated symbol name, and returns the address of where the symbol is loaded in memory. If the symbol is not found, dlsym() returns NULL. dlsym() searches for the symbol using the breadth-first search algorithm through the library dependency tree. Since a valid value of one of the symbols returned by dlsym() could possibly be NULL, the correct way to test for errors with dlsym() is to call dlerror() to clear old error conditions, then call dlsym(), and then call dlerror() once more.

Two special pseudo-handles are defined for use. These pseudo-handles are called RTLD\_DEFAULT and RTLD\_NEXT. The former finds the first occurrence of the desired symbol using the default breadth-first search algorithm through the library dependency tree. The latter will search for the second occurrence of the same symbol. This is useful if you know that there are two libraries which contain the same symbol name, and you know that the symbol ought to be loaded from the second library.

dlclose: int dlclose(void \*handle)

dlclose() decrements the reference count on the dynamic library handle specified in the parameter handle. If the reference count drops to zero and no other loaded libraries use symbols in it, then the dynamic library is unloaded. dlclose() returns 0 on success and non-zero on failure.

GCC Function Attributes:

We’ve already noted the existence of #pragma directives. As far as I can tell, #pragma directives are moreso supported on the Microsoft Visual Studio Compiler (MVSC). Although GCC also supports #pragma directives, GCC has its own unique commands called function attributes for modifying the attributes of a function. It is pretty bad practice to use GCC attributes because they are not portable whatsoever, but I would like to highlight them for exposures sake. GCC has a special decorator: \_\_attribute\_\_ (()) which is used to accept arguments that modify the behavior of how the compiler treats the function declared within the same statement. Arguably one of the most commonly used ones is \_\_attribute\_\_ ((inline)), which forces the compiler to inline the function that proceeds it. Another example is deprecated, which will cause the compiler to produce a warning if the function is invoked in any of the program’s source files. There are many many function attributes, the breadth of which I could not possibly cover. There are arguments as specific as telling the compiler to create SIMD versions of a function. For more information on GCC function attributes, see GCC’s website: <https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html>.

Assembly Programming in C:

Did you know that C is backwards compatible with Assembly? It’s true, with the help of the \_\_asm\_\_ keyword, we can define a code block for assembly code. Alternatively, GCC has a compiler extension keyword – asm (without underscrores), however, the syntax for this is a bit different and will only compile under GCC. \_\_asm\_\_ also works even for code compiled with -fno-asm (this flag blocks the use of the asm keyword, not \_\_asm\_\_). There are two forms that we can use to write assembly: basic asm, and extended asm. Using extended asm typically produces smaller, safer, and more efficient code and provides the easier method of accessing data from external C code. There are a few use-cases where basic asm may be a consideration:

1. Extended asm statements must reside within a C function, so if your goal is to write inline assembly at the file scope (outside of C functions), you must use basic asm.

2. Functions declared with the naked attribute (we’ll look at attributes in a moment) require basic asm.

3. You must store a C variable in a specific register. We covered the register keyword, which can hint to the compiler that we want to put a variable in one of the CPU’s registers. In order to force the compiler to place a variable in a specific register, we can use a statement such as the following: volatile register int x asm(“eax”); This forces the compiler to turn off compiler optimizations for this portion of code and place x in the register eax.

We use the following format to declare the start of basic asm:

\_\_asm\_\_ [asm-qualifiers] (

assembler-instructions

)

Extended asm uses the same format, but with more optional arguments:

\_\_asm\_\_ [asm-qualifiers] (

assembler-template

[: output-operands]

[: input-operands]

[: clobbers]

[: goto-labels]

)

Qualifiers can be one or more of the following (separated by spaces):

**volatile:** Disable branch predicting or other optimizations that may produce side-effects.

**inline:** Make the asm code as small as possible.

**goto:** Informs the compiler that the asm statement may perform a jump to one of the labels listed in the goto-labels section.

Let’s look at each of the parameters for the extended asm format:

**assembler-template:** The assembler template is a C string that contains the assembler instructions. Unlike basic asm, the assembler template requires us to manually include newlines and tabs in our assembly code (with use of the ‘\n’ and ‘\t’ characters). This is because it is just one large string that will be expanded into a file for execution by the compiler in a later stage of compilation.

**output-operands:** A comma-separated list of the C variables modified by the instructions in the assembler template. Empty lists are permitted.

**input-operands:** A comma-separated list of C expressions read by the instructions in the assembler template. Empty lists are permitted.

**clobbers:** A comma-separated list of registers or other values changed by the assembler template. Empty lists are permitted.

**goto-labels:** A comma-separated list of goto labels which exist in the C code that the assembler template is able to jump to.

Note: The total number of input + output + goto operands is limited to 30.

Here is an example program that uses extended asm. It copies the C variable src into the C variable dst and then adds 1 to dst:

#include <stdio.h>

int main(void) {

int src = 1;

int dst;

\_\_asm\_\_ (

“movl %1, %0\n\t”

“addl $1, %0”

: “=r” (dst)

: “r” (src)

);

printf(“%d\n”, dst);

return 0;

}

Likely the most puzzling aspect of this code are the “=r” and “r” symbols. These are constraints for our C variables. The equals sign is one of four optional modifiers, the others being the plus sign, ampersand, and modulo/percentile symbols. Here is what each refers to:

**=:** The following qualifier is an output value i.e., the value is expected to be replaced

**+:** The operand is read from and/or written to by an instruction

**&:** The register used to store the operand must be an output only register, and may not serve a secondary purpose as an input register

**%:** The operands may be commutative (changing the order of operands does not affect the result)

The ‘r’ itself is a qualifier that denotes that the value will be stored in a General Purpose Register (GPR). There are several other letters, some of which only make sense when combined, and each of which have their own meanings. As mentioned, ‘r’ denotes a GPR, and it lets the compiler decide whichever register it thinks is best suited for the data. This does not always work in our favor, however, as the compiler may clobber existing registers that we’re already using. For this reason, we have the a, b, c, and d letters to hint to the compiler that we’d like a variable to be placed in the rax, rbx, rcx, or rdx register (these can be combined to give the compiler a pool of registers to choose from). Alongside a, b, c and d, there is also S and D, which are the rsi and rdi registers. The letter ‘m’ can be used alongside ‘r’ to specify that data may be stored in either a GRP or in memory.

Let’s look at an example where we invoke the sys\_write syscall to print a string to stdout. A very useful resource which I recommend bookmarking in your browser or downloading is the Linux system call table for x86 64, which can be found here: <https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/>. I’ll paste my completed example and then we’ll walk through it:

#include <stdio.h>

#include <string.h>

void run(void) {

const char \*msg = "Hello from asm\n";

size\_t len = strlen(msg) + 1;

\_\_asm\_\_ (

"movq $0x01, %%rax\n\t" /\* 0x01 is the sys\_write system call \*/

"movq $0x01, %%rdi\n\t" /\* 0x01 is the stdout file descriptor \*/

"leaq (%q0, 1), %%rsi\n\t" /\* Load effective address of msg \*/

"movq %q1, %%rdx\n\t" /\* Load len \*/

"syscall" /\* Invoke sys\_write call \*/

: /\* No output values \*/

: "b" (msg), "c" (len) /\* Suggest storing msg in rbx and len in rcx \*/

);

}

int main(void) {

run();

return 0;

}

Most of what’s happening here is fairly self-explanatory, but don’t worry, I’ll cover it line by line. As we read through, keep in mind that this is AT&T syntax (which is the default on Unix-like systems). This means that variables are transfered from left to right instead of right to left. Since registers begin with ‘%’ in AT&T, we need to escape the character with a secondary ‘%’, since GCC interprets the modulo symbol as a special symbol. If you’re not familiar with AT&T syntax, note that the mnemonics such as mov and lea are appended by ‘q’ in this case, which stands for “quadword” i.e. 64 bits on x86. Other postfix operators include ‘b’ for byte, ‘s’ for single, ‘w’ for word, ‘l’ for long, and ‘t’ for “ten bytes”. The postfix operator should correlate with the size of your registers and your data. Anyhow, looking at the x86 ABI in the reference provided previously, we see that the sys\_write system call is associated with decimal 1, so we move the absolute value into rax. Next, rdi takes the file descriptor that we want to write to, which in our case is stdout (if you know a bit of POSIX, you’re probably familiar with the fact that stdout always corresponds to fd 1). Next we want to store msg in rsi, however, rsi takes in a const char\* as the argument. A simple mov instruction will not suffice; instead, we must load the effective address of our msg variable. Note that both the msg and len variables are registered as input operands. Instead of using “r” to allow them to be stored in whichever GPR the compiler decides, I chose to suggest msg be placed in rbx, and len be placed in rcx. The reasoning for this is because rax, rdi, rsi, and rdx are all required for the system call to complete successfully, so rbx and rcx are the only remaining registers for msg and len to be loaded into initially that wont clobber existing data (we’ll look at an optimization that we can do here in a bit). The lea instruction is perhaps the most confusing. This is an instruction which takes multiple parameters for the first operand, denoted by the brackets. We load our 0th parameter (msg) using %q0. The ‘q’ is not required here, but it explicitly denotes that we want to store the variable as a quadword. The 1 indicates the multiplier for the offset of this variable. We don’t require an offset, so we just multiply by 1. Finally, we move our 1st parameter (len) into rdx using %q1. And of course, we need to run the syscall instruction to have it actually execute. We can compile this code as normal with GCC. Assuming the source file is named syswrite.c, we can compile with gcc syswrite.c -o syswrite. The best way to debug the code is by compiling with the -S flag e.g., gcc syswrite.c -S syswrite.s. Looking at this file, we can indeed see the instructions that were generated:

movq $0x01, %rax

movq $0x01, %rdi

leaq (%rbx, 1), %rsi

movq %rcx, %rdx

syscall

You may hint to the compiler the type of assembly that is used in your code by using the -masm flag (MASM being the assembler that gcc uses). The typical options that are assigned are either att or intel e.g., -masm=intel. If you write code written in AT&T syntax, but compile using intel as the compiler argument, the compiler will throw a compile time error and point to the instructions that it is not able to interpret. If you’re just trying to debug someone elses code and want to change the assembly flavor in gdb, you can enter *set disassembly-flavor (att | intel)*. Compiling the same code with gcc syswrite.c -masm=intel -S syswrite.s will yield the following instructions:

mov $0x01, rax

mov $0x01, rdi

lea (rbx, 1), rsi

mov rcx, rdx

syscall

Note that this code fails to assemble, because in Intel syntax expects operands to flow from right to left. In the rare case that you write assembly code which contains only commutative (i.e. reversable) instructions, there is a special syntax that we can use. Take this code for example:

xchg{l} {%%}ebx, %1

We can use the {} braces to specify characters that should only be present when using AT&T syntax. Since the xchg instruction just swaps the two operands, their order does not matter. This code will assemble successfully for both AT&T and Intel syntax.

Now let’s finally look at that optimization in our previous code example that I promised. Recall that I said we specified “b” and “c” to place our msg variable in rbx and our len variable in rcx. Well, instead of doing that, we can actually just place them into our parameters directly, saving a few instructions. The code can be reduced to:

\_\_asm\_\_ (

"mov{q} $0x01, {%%}rax\n\t"

"mov{q} $0x01, {%%}rdi\n\t"

"syscall"

:

: "S" (msg), "d" (len)

);

Note the use of S to place msg into rsi and d to place len into rdx. These are the buf and count parameters specified by the x86 ABI for the sys\_write syscall.

I should also mention the fact that we can use labels to capture our input and output operands. When using multiple variables, it can become quite cumbersome to keep track of their indices with %0, %1, %2, etc... We can define a label for our variable using square brackets like so:

: [label] “r” (variable)

Then we can reference this variable using %[label] instead of %n.

Compiler Intrinsics/Builtins:

Most compilers have a feature known as compiler intrinsics (sometimes called builtins). The idea is that the compiler contains functions which share the same symbol names as the ones in the C runtime library but are implemented in assembly and can be inlined. Some compiler intrinsics are portable, and some are not. Function intrinsics in gcc will begin with the \_\_builtin\_ prefix e.g. \_\_builtin\_csin(). The list of functions which have builtin alternatives can be found on gcc’s website: <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html> . There is an option in gcc to disable compiler intrinsics using -fno-builtin. I generally use this flag for debug builds, and omit it for the release build.

Libraries:

Now that you’ve read this document and are officially an expert at writing C, it’s time to become accustomed to some of my personal favorite libraries written for C. Since C does not have any sort of package manager, and therefore no global repository for querying specific libraries, it is often difficult to find trusted ones that are actually useful. I’d like to bring a couple to your attention to play around with:

**- Raylib:** A very intuitive and lightweight wrapper around OpenGL.

**- Lua:** This ones not actually a library, but did you know that you can actually just build Lua from source and get access to APIs for interfacing with the Lua scripting language? This is useful for hot-reloading code by binding C variables to the Lua stack and updating them from a Lua script.

**- Ncurses:** By far the best terminal manipulation library for C. Used in pretty much every TUI application.

**- FFMPEG:** A widely-used video and audio codec library.

**- Musl:** A lighter version of the C standard library (compatible with POSIX 2008 and C11); designed for static linking.

**- Libcsv:** A simple CSV parser.

**- Check:** There are a ton of testing libraries for C, but I’ve always used check for writing unit tests.

**- Zlib:** A compression/decompression library written by the same folks that create the PNG file format!

Of course, my list is extremely tiny and heavily cherry picked, but if you want a more extensive list, I recommend checking out this person’s currated list of C libraries and tools: <https://github.com/oz123/awesome-c>.