The C Programming Language

Neil Kingdom

October 1, 2024

Contents

1 Abstract

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!

2 Introduction

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).

3 Data Types

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.

3.1 Void

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).

3.2 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).

3.3 Optional Reading (Advanced): Character Control Sequences

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:

Keyboard Sequence	ASCII Character	Hexadecimal	Semantic Meaning
ESC D	IND	0x84	Represents an index
ESC E	NEL	0x85	Represents the next line
ESC X	SOS	0x98	Represents the start of a string
ESC [	CSI	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 specification, 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. Another resource which I found helpful can be found here.

3.4 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
  - Binary literal (base 2): Begins with '0b' e.g. 0b00001111, 0b01010101. Please note that binary literals are only supported by some compilers and are not generally portable.
  - Octal literal (base 8): Begins with '0' e.g. 0777, 0123, 0655
  - Hex literal (base 16): Begins with '0x' e.g. 0xFFFF, 0xC0FFEE, 0x01
- **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.

3.5 Short

short is short form for ”short int” (say that 5 times fast). 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, the following lines are equivallent:

```
short shrt = 10000;
short int shrt = 10000;
```

Same as int, a suffix of ‘u’ or ‘U’ to specify that it is an unsigned short.

3.6 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 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and for unsigned, it is 0 to 18,446,744,073,709,551,615! long has its own dedicated suffix, which is ‘l’ or ‘L’. This can be used in conjunction with ‘u’ and ‘U’. For example, the following line is valid:

```
unsigned long l = 100000000ul;
```

3.7 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;
```

3.8 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.

3.9 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, and for programs which utilize a lot of maths, will end up saving you a lot of program space.

3.10 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.

4 Creating Aliases With 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. In this example, ull_t is a typedef (alias) for ”unsigned long long int”. Typedefs are never strictly necessary, but you will innevitably come accross them as you use other people’s libraries or APIs. A typedef can be declared almost anywhere in your program. Conventionally, typedefs will end with the suffix ”_t”, though this is a matter of preference.

5 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.

5.1 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 I’ve lumped it in here because it functions the same as a typedef. 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 dealing with lots of boolean logic and want to make your code more legible, or when you want to return true or false from a function rather than using a *sentinel* value such as 1 or 0.

5.2 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 that you understand that these typedefs guarantee the specified number of bits, regardless of platform/architecture.

5.3 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.

5.4 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:

```c
#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.

6 A Word on Man Pages

As a piece of useful advice, you can search for header files within the Linux man pages to get more information regarding function or typedef declarations. For example, to find out more information about bool, you can execute the command "man stdbool.h" in your terminal, which will provide you with a list of functions and typedefs that the stdbool.h header file defines. For headers which exist within a directory e.g. <sys/-something.h> you can usually replace the '/' with an underscore e.g. "man sys_something.h". Sometimes C APIs share a name with the Linux command. For example, stat is a command, whilst simultaneously being a C function. In order to search for the C reference, do "man 3 stat". The man command has 8 separate categories that it divides information into, so explicitly selecting the number 3 will get you the C reference, whereas 1 is reserved for shell commands. If you'd like to know more about man section, see my Linux document or read the man page for the man command!

7 Preprocessor Directives

I will now 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 that way for now, but there is one thing that I must quickly explain in layman's terms. The compiler has several 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:** Undefines a preprocessor macro.
- **#if:** An conditional statement ran during the preprocessing phase.
- **#elif:** Equivallent to an else-if statement, but ran during the preprocessing phase.

- **#else:** Equivallent to an else statement, but ran during the preprocessing phase.
- **#endif:** Terminates a conditional #if block.
- **#ifdef:** Short form for #if defined(x).
- **#ifndef:** Short form for #if !defined(x).
- **#error:** Prints an error message to stderr during the preprocessing phase.
- **#pragma:** Issues special commands to the compiler during the preprocessing phase.

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

7.1 #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 our own opening and closing curly braces (please don't do this)!!!

```
#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.