The C Programming Langauge

Neil Kingdom

June 26, 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).