The Bloody Way to C

A Brutalist Approach to C Language—No Boilerplate

Simone Lungarella (simonelungarella@gmail.com)

2025-05-11

Contents

Preface

This e-book is a *humble* attempt to describe *C* language while actively trying to learn it. I enjoy writing code and technical documentation and I decided to produce this guide under MIT licence. It is *not* intended to be fully comprehensive and complete, it only contains what I've learned and follows my very personal style.

It will be consistently updated and improved until completion and kept—as much as possible—accessible.

To understand every aspect of *C*, many tools will be used and all examples will refer to CLI commands. I will be using Neovim as text editor and operate on a Linux machine. The output of commands and all examples may differ from machine to machine but the concepts will hopefully remain valid.

I strongly believe that the best way to develop software is by using *CLI* and lightweight text editors such as `neovim` or `vim`. Whenever is possible I will avoid using browsers to search for documentation by preferring usage of `man` directly into the terminal. This will keep low the friction and avoid the necessity to leave the home row of my keyboard.

Introduction

The C language, as an idea of computation

C was invented in Bell Labs when Ken Thompson was working on Unix. Following the idea that a good operating system should have had a high level compiled language. After abandoning the first attempt on creating a compiler for `Fortran`, a smaller new language was created and named `B`. `B` better fitted P2P11 but was not enough to port Unix from Assembly. `C` was created with a set of feature that were missing in `B` and was a much better fit for the Unix system.

C was a better language mainly because its multiple distinct types:

- pointers;
- integer;
- floating point numbers: float;

In that sense, *C* language can be visualized as *B* with types where all types can also be imagined as integers since pointers—in very simple terms—are integers and so are structures. In fact, structures are a set of integers representing offsets of each field position in memory and values of the very same fields. This simplicity can be considered the strength of the language as it can be easily picked up by new developers, layered to build a powerful abstraction and, with that, imagine in simple terms complex topics and algorithms.

1 Anatomy of a C program

1.1 Execute a C program

C is a compiled language, this means you cannot execute a file containing the main function. It requires to be compiled.

You can use: `cc` to compile a *C* program. `cc` is a Unix command that let you easily communicate with the compiler. You can use: `cc --version` to check what compiler does it use.

```
cc (GCC) 14.2.1 20240912 (Red Hat 14.2.1-3)
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Let's consider the following simple *C* program contained in a file named—for instance—*hello_world.c* (how original):

```c
#include <stdio.h>

int main(void) {
  printf("Hello World\n");
  return 0;
}
```

You can use: `cc hello_world.c` to compile it to an executable program.

The compiler generates an executable binary file named: *a.out*. This file is executable and runs your program. You can use: `file a.out` to check information about the generated file.

```
./a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, \
interpreter /lib64/ld-linux-x86-64.so.2, \
BuildID[sha1]=d589730d718a032a35f848fe8d280063a6cee18c, \
for GNU/Linux 3.2.0, not stripped
```

If you want to check the content of the generated binary file, you can use: `hexdump -C a.out`.

You can also generate Assembly code using: `cc -S hello_world.c` if the compiler supports this feature.

```
    .file   "hello_world.c"
    .text
    .section    .rodata
.LC0:
    .string "Hello World"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (GNU) 14.2.1 20240912 (Red Hat 14.2.1-3)"
    .section    .note.GNU-stack,"",@progbits
```

With a given compiler, you can tweak many compilation aspects. For instance, `cc -O2 hello_world.c` tells the compiler to optimize the generated executable. A more optimized version of the executable is also slower to be generated and, if that does not make much sense for small programs, it can output a much better version of the program when a high enough level of complexity has been reached.

With GCC compiler, you can see that our simple program make use of `puts` syscall, however, this depends on the compiler itself and, often, with different compilers, the line: `printf("Hello World\n");` is compiled using `printf` syscall instead.

Using `-O2` flag can make the compiler use `puts` as this syscall is faster than `printf`. This is a simple, yet meaningful, example but in such a small program it does make no difference in terms of execution speed. The compiler is very good at improving written programs if given enough time. While developing, though, a low compilation time is often preferred.

> You can check the standard C library from the terminal using `man` or `--help` flags. For example, you can use `man 3 puts` or `man 3 printf` to check documentation of both syscalls (3 makes sure to output the C library description).

1.2 Include other source code

In the very first line of our simple program, you can see a preprocessor directive. This line simply tells to the compiler that a file need to be included into the program. The compiler, before the compilation, take the content of the file and *paste* it at the location. In this case, `<stdio.h>` declares the prototype of `printf` function so to instruct the compiler on how to execute that specific call. To prove this point, you can remove the first line and replace it with: `int printf(const char *restrict format, ...);` which is the prototype of the function we want to call.

```c
int printf(const char *restrict format, ...);

int main(void) {
  printf("Hello World\n");
  return 0;
}
```

`#include` can also be used to include other *C* files. In fact, you can move a single line to a different file and than compile a program that includes the file on the line you want it to be replaced.

```c
#include <stdio.h>

int main(void) {
  #include "file.c"
  return 0;
}
```

The generated assembly or machine code will be equivalent.

1.3 Functions

This very simple program has a single function named `main`. A function has always a return type, an *optional* list of parameters, and a body. The signature of the function `main` has a return type specified as `int` —this means that the function must return an integer value.

Parameters are defined inside the brackets of the function and they too have a specific type. It is also possible to define a function that does not require any parameter. This can be explicit, using `void` as the function `main` does, or implicit, by simply avoiding specifying any parameter: `int main() {}`.

Functions can call other functions too!

```c
#include <stdio.h>

int sum(int a, int b) {
  return a + b;
}

int main(void) {
  printf("Hello World %d\n", sum(10, 20));
  return 0;
}
```

The function `main` is a special function, in fact, it is the only function that is automatically called by the program. Other functions must be explicitly called. This means that a valid *C* program must define the `main` function.

1.4 Variables

Functions parameters are variables existing only during the function execution. There are variables which are not involved only in function calls but also have a meaning in the callee context or even in the whole program context.

1.4.1 Scope

Variables can have different scope. In the previous example, the function `int sum(int a, int b)` has two variables as parameters having a local scope. When variables are local, they are valid only within the function context and have no meaning to other functions.

To understand this concept, let's consider the following program:

```c
#include <stdio.h>

int sum(int a, int b) {
  return a + b;
}

int main(void) {
  int a = 10;
  int c = 20;

  printf("Hello World %d\n", sum(a, c));
  return 0;
}
```

This is a valid *C* program, equivalent to the previous, and, as you can see, the variable named `a` exists twice with the same name. This is possible because in both cases, the variable scope is local to the function itself and it's removed after the function has returned its value.

> The function `main` has a return type but since it is automatically called by the program, the only one that can be interested in its value is the callee: the program executor. If executed from a shell, the program returns its value and can be shown with `./a.out; echo $?`. This is quite useful combined with the fact that `0` is equivalent to `true` in Unix shells.

Variables can also have a global scope. A global variable is seen by every function and initialized only once.

```c
#include <stdio.h>

int x = 0;

void incr(void) {
  x = x + 1;
  printf("%d\n", x);
}

int main(void) {
  incr();
  incr();

  return 0;
}
```

In such cases the value of x is incremented by one each time the function is called. Values of local variables can also be retained through multiple function calls if they are defined as static: `static int x = 0;`.

It's important to highlight that, in *C*, variables are passed *by value*. This means that whenever a function is called, it cannot modify any existing variable local to the callee but, for each of its parameters, a copy of the value is passed. To modify local variables with functions it is necessary to use *pointers* which will be described extensively in the next paragraph.

1.4.2 Type

We have seen variables having type *int*, but there are multiple primitive types that can define different kinds of data. For simplicity, a subset of common primitive types is shown in the following table, refer to the standard documentation to explore all different existing types.

Type	Common size (b)	Description
int	32	Signed integer numbers
float	32	Floating point numbers
double	64	High precision floating point numbers
char	8	Characters
short	16	Shorter signed integer numbers

> All size reported are not guaranteed by C specification, it mainly depends on architectures.

In many cases, types are automatically *promoted* to a higher size type to easily handle similar cases. For instance, `printf` will promote `char` or `short` values to `int` enabling developers to simplify the usage of the function.

```c
short s = 400;

// `s` is automatically converted.
printf("%d\n", s);
```

This happens with functions such as `printf`, which accept a variable number of parameters (variadic function), but also during expressions evaluations if necessary.

```c
char c = 127;

// Before evaluation, `c` is promoted to int.
int i = c + 1;
printf("%d\n", i);
```

Since the size of types is variable and depends on the architecture, there is a specific function that returns the size of a specific variable: `sizeof(var)`.