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 it 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

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 Execute a C program

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

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* 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).

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.

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.

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.

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 a dedicated chapter.

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 into 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)`.

Variables can represent a single value or a collection of values. To define a variable and store multiple values of the same type, *C* provides **Arrays**.

```c
int array[5] = {1, 2, 3, 4, 5};

printf("%d\n", array[0]);
```

Arrays can store multiple values in different positions and track a contiguous block of memory. To access value in a specific position you can use each index starting from `0` up to `n-1`.

Arrays of characters are named *strings*, and—since they are very common—there is a simpler way to initialize them:

```c
char phrase[] = "Hello World";

printf("%s\n", phrase);
```

It is not mandatory to set the size of the array as the compiler will automatically do it by checking the its size. You can evaluate the size of a string too using: `sizeof(string)`, which is also evaluated at compile time.

> You will notice that the size of strings, returned by `sizeof`, is always greater than the amount of character in the string. Strings always require the null terminator: `\0` that tells the program when the array ends and initializing a string using quotes automatically adds the null terminator.

5 Code blocks

Code blocks are blocks delimited by brackets that can be part of functions. Each function has at least one code block. Variables declared in a specific block have a local meaning and occupy a different memory block.

```c
#include <stdio.h>

int main(void) {
    int i = 5;

    {
        int i = 3;

        // (4 bytes) stored at 0x7fff7cae44b8
        printf("(%zu bytes) stored at %p\n", sizeof(i), &i);
    }

    // (4 bytes) stored at 0x7fff7cae44bc
    printf("(%zu bytes) stored at %p\n", sizeof(i), &i);
}
```

> `&i` returns the memory address where the variable is stored, more about *pointers* in the following chapters.

This simple program will show how the two variables, having the same name, will be stored in two consecutive memory blocks that differ by exactly 4 bytes (from 0x[...]8 to 0x[...]c).

5.1 Conditional code blocks

Often, the linear execution of a program needs to be interrupted to take a direction based on a specific condition. Conditional code blocks are blocks of code executed only if a specific requirement is satisfied.

The keyword `if` defines a conditional block and the condition that needs to be met for the execution:

```c
#include <stdio.h>

int main(void) {
    int i = 5;

    if (i> 3) {
        printf("Value %d is greater than 3\n", i);
    } else {
        printf("Value %d is not greater than 3\n", i);
    }
}
```

Conditional blocks are optionally enhanced with `else` or multiple `else if` constructs that build up the logic based on multiple different conditions.

> When conditional code blocks are constituted by a single statement, brackets are optional: `if (i > 0) printf(i);` .

5.2 Switch

Another useful way to handle conditional blocks is by using `switch` keyword.

```c
#include <stdio.h>

int main(void) {
    int i = 5;

    switch(i) {
        case 5:
            printf("It is a five!");
            break;
        case 3:
            printf("It's not a five.");
            break;
        default:
            printf("No case matched.");
            break;
    }
}
```

Switch blocks can be used to execute code based on a matching condition in an elegant way. Each case code block have to be terminated with the `break` keyword to avoid executing following code blocks too.

6 Iterating

To be Turing-complete, a language must have some kind of looping logic. *C* has many way to iterate the execution of a code block: `for`, `while`, `do-while`. Loops let the program jump at the start of a code block for multiple iteration each time the condition is met. A way to achieve the same result is by using the keyword `goto`.

The keyword `goto` interrupt the program execution and start from a specified *label*.

```c
#include <stdio.h>

int main(void) {
    int i = 0;

again:
    printf("%d", i);
    i++;
    if (i < 10) goto again;
}
```

In the example, when the condition is met, the instruction `goto` make the program jump back to the line under the specified label.

The logic to iterate a code block or a set of instruction can be also written by using a *while loop*.

```c
#include <stdio.h>

int main(void) {
    int i = 0;

    while(i < 10) {
        printf("%d", i);
        i++;
    }
}
```

6.1 Recursion

Another way to execute multiple times a specific code block, is by using recursion. We talk about recursion when a function call itself. In the following example, `count` is a recursive function.

```c
#include <stdio.h>

void count(int start, int end) {
    if (start > end) return;
    printf("%d\n", start);
    count(start + 1, end);
}

int main(void) {
```

```
    count(0, 9);
    return 0;
}
```

Calling multiple time the same functions is equivalent to have multiple code blocks and—as said—each code block instantiate its variables in different memory addresses. This means that recursion, by instantiating multiple time the same variables, uses more memory than a simple while loop.

The recursive function *count* can be written using `goto` instead of recursion.

```
void count(int start, int end) {
    if (start > end) return;
iterate:
    printf("%d\n", start);
    start++;
    goto iterate;
}
```

6.2 For loops

Another way to iterate over elements is by using a *for loop*.

```
#include <stdio.h>

int main(void) {
    for(int i = 0; i < 10; i++) {
    printf("%d", i);
    }
}
```

This kind of loop defines the starting value of the loop counter, the exit condition and the increment rule on the same line, then defines the code to be executed in a code block. Loop counter, exit condition and increment rule can be omitted and handled manually if necessary.

```
#include <stdio.h>

int main(void) {
    int i = 0;

    for(;;) {
        if (i < 10) break;
        printf("%d", i);
        i++;
    }
}
```

> It's important to notice that if the loop counter is defined the same time it is initialized, the variable will be local to the loop block.

7 Pointers

Pointers are special variables that indicate an area of the memory of a specific type. They are declared with an asterisk as in the expression: `int *y;` where `int` denotes the type of the data allocated at the address. Pointers can map an address of any type and its size depends on the machine's address size.

Every variable has its own address in memory, to get the address of a given variable, can be used the operator: `&`.

```c
#include <stdio.h>

int main(void) {
    int x = 5;
    int *y = &x;

    printf("x is stored at the address: %p\n", y);
    return 0;
}
```

Values of variables can be modified interacting with its pointer too, so `*y = 10` would alter the value of the variable `x` if the variable is declared as equal to the address of x: `int *y = &x`.

Pointers are extremely powerful because they make possible updating variables without any extra memory allocation. It is possible, in fact, to update a variable by calling a function that does not instantiate any local variable but only access and manipulate the value using its pointer.

```c
#include <stdio.h>

void incr(int *p) {
    *p = *p + 1;
}

int main(void) {
    int x = 5;
    int *y = &x;

    printf("x was: %d\n", x);

    // This instruction alters the value of x using its pointer
    incr(y);
    printf("x now is: %d\n", x);
    return 0;
}
```

> Pointers have always the size of a `intptr_t` as it represents a memory address. The type declared with the pointer indicates the type of the data stored at the given address.

7.1 Pointer's math

To access the value stored in a memory address defined by a pointer, it is necessary to dereference it using `*`.

```c
#include <stdio.h>

int main(void) {
    int x = 5;
    int *y = &x;

    printf("x is: %d\n", *y);
    return 0;
}
```

However, such operation, can also be done using the form: `y[0]` which is commonly used for arrays. This type of representation of the values stored in specific addresses is very useful when combined with the fact that, pointers, follow a very specific math and their index can be incremented to access contiguous memory addresses.

```c
#include <stdio.h>

int main(void) {
    char x[] = "Hello";
    // In C, array's names are also pointers.
    char *p = x;

    printf("At second position value is %c\n", p[1]);
    printf("At third position value is %c\n", *(p + 2));

    return 0;
}
```

Accessing data in that way works with all types—the language will handle the incrementation accounting for different types by incrementing the address by the given index multiplied by `sizeof(<type>)`.

The fact that pointers are similar to arrays, combined with the fact that operations are much simplified on pointers, is very powerful and give developers much expression strength.

```c
#include <stdio.h>

int main(void) {
    char x[] = "Hello";
    char *p = x;

    while(*p != 0) {
        putchar(*p);
        p++;
    }

    return 0;
}
```

Pointers are variables, and as such, they are stored in some address in memory. This make it possible to have pointers to pointers: `int **z = &y;` where `y` is a pointer to an integer.

> For a comprehensive program that make use of all concepts introduced, check out Conway's Game of Life.

7.2 Working with memory

In *C*, memory is handled manually by the developers. This is a very powerful feature of the language but make it very easily error-prone.

We use `malloc()` to allocate part of the memory and `free()` to release it.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    char *mystr = malloc(11);
    mystr = "Hello World";

    printf("%s", mystr);
    free(mystr);
    return 0;
}
```

For a better use case, you can check Prefixed Strings.

7.3 Function pointers

As variables can be referenced by pointers, so are functions. In fact, functions occupy space in memory likewise variables.

A function pointer is declared by writing the function's return type, followed by (`*pointer_name`) and the parameter list, for example: `int (*fptr)(void);` .

```c
#include <stdio.h>

int main(void) {
    int (*fptr)(void);
    fptr = main;

    printf("%p\n", fptr);

    // The following instruction effectively calls the main method
    fptr();
    return 0;
}
```

> In this example, calling `main()` recursively (via the function pointer) will eventually cause a stack overflow, leading to a segmentation fault.

Function pointers are used, for example, by many sorting functions. The `qsort()` function, for instance, requires a pointer to a *comparator function* that abstracts the comparison logic, making it possible to use the same sorting implementation with user-defined comparison functions. This allows the sorting algorithm to be applied to any type of data structure.

8 Structures

When working on complex scenarios, built-in *C* types are often not enough. In *C*, it is possible to define structures having specific fields using the keyword `struct`.

```c
struct item {
    int value;
    char *name;
};
```

Structure fields can be accessed using the operator: `.` (dot) when locally allocated and the operator: `->` (arrow) when handling structure pointers. Fields can be read and wrote using the same operators, structure can greatly improve code readability and simplify the software.

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

struct item {
    int value;
    char *name;
};

int main(void) {
    struct item *building = malloc(sizeof(struct item));

    // Alternative ways of instantiating structures in C
    // struct item building = {.value = 10000, .name = "House"};
    // struct item building = {10000, "House"};

    building->name = "House";
    building->value = 10000;

    printf("Building: %s has the value of: %d", building->name, building->value);

    free(building);
    return 0;
}
```

Structures can be defined as part of the same or other structures. To see an example of structure usage, check out Tac.

> Using different operators to access structure fields helps make the developer's intention clear. While the compiler can already distinguish whether a variable is a pointer or not, explicitly choosing the appropriate operator improves code readability.

8.1 Memory management

Structures, by default, guarantee that each member is stored in a memory address multiple of its size. This means that field order has an impact on memory usage. For example, a structure having three fields: `int, char, int` will make use of 12 bytes by adding a 3-bytes padding after the char.

It is not common but it is also possible to instantiate structures in functions and return them as values. This approach does not require usage of `malloc` and `free` but with heavy structures it can easily decrease the performances of the software since all the bytes need to be copied.

8.2 Structures as user defined types

All *C* types have a specific name and it must be defined whenever a new variable is declared. However built-in types can also have user-defined aliases. To define an alias for a type, the keyword `typedef` can be used.

```c
typedef int errorcode;

int main(void) {
    errorcode x = 20;
    return 0;
}
```

This is a convenient way to simplify structure usage and avoid explicitly declare a variable of a user-defined type using the keyword `struct`. This approach can be often seen when working with libraries, for example, one way to read a file is: `FILE *fp = fopen("file.txt", "r");` where `FILE` is a structure defined by the library that contains multiple useful information about the file.

```c
#include <stdio.h>

typedef struct {
    int numerator;
    int denominator;
} fract;

// `typedef` can also be used on user-defined types
typedef fract *fractptr;

int main(void) {
    fract f;
    f.numerator = 10;
    f.denominator = 20;

    printf("%d/%d", f.numerator, f.denominator);
    return 0;
}
```

> Note that when using `typedef` with structures, the defined name follows the structure itself and not the `struct` keyword.

8.3 Unions

Structures can be defined with multiple fields starting at the same offset. Unions are specific sections of structures where multiple fields are defined and share the same memory. This can be useful, for example, when multiple values are mutually exclusive; check Redis expr.c for a better use-case example.

```
struct foo {
    union {
        int i;
        unsigned char a[4];
    };
};
```

If multiple unions are present, it is convenient to assign a name to each specific union.

8.4 Bit fields

It is possible, in *C*, to define structure fields with a specific bit size. Assigning greater values in such fields provoke a wrapping around if they are defined as `unsigned`, otherwise the behaviour is **undefined**.

```
struct foo {
    unsigned char a: 4;
    unsigned char b: 4;
    unsigned char c: 8;
};
```

9 System calls

Most Unix-like systems implement similar system calls, but POSIX standardizes the higher-level APIs rather than the system calls themselves. Many of these systems are largely POSIX compliant, which makes *C* programs portable across multiple platforms.

Programs can rely on system calls but often the complexity is abstracted behind a library. For example, to read a file you can use `fopen` from the `stdio.h` or `open` from `fcntl.h` which is part of the POSIX API.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
    int fd = open("file.txt", O_RDONLY);
    if (fd == -1) {
        perror("Unable to open file");
        return 1;
    }
    close(fd);
    return 0;
}
```

> To learn about the interface of system calls you can check the second section of man pages.

The abstraction layer of a library simplifies the use of system calls; however, its design typically optimizes for common interaction patterns. When more specialized behavior is needed, writing custom functions that invoke system calls directly is often more performant. For instance, functions like `printf` use internal buffers to improve efficiency, but this also means developers lose fine-grained control over when data is actually written to the system.

System calls are the primary mechanism by which processes communicate with the kernel. However, since processes can also manipulate memory, there is another way to interact with the kernel: memory mapping. Using `mmap`, a process can map a file or device into its address space and access its contents directly through a pointer. In such cases, the process can reduce the number of system calls and access the file more efficiently.