

Elements of Operating Systems

Chapter 4

In the Beginning Was C

J. Fornés M. Farreras Y. Becerra

Barcelona School of Informatics
Universitat Politècnica de Catalunya

Bachelor's Degree in Informatics Engineering — Operating Systems

Preview for readers · September 28, 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

This text was written with the help of Claude, Anthropic's language model. Any errors that remain are the authors'.

4 In the Beginning Was C

C is not a big language, and it is not well served by a big book.

— Brian Kernighan and Dennis Ritchie,
The C Programming Language [1]

This chapter will not teach you to program. You already know how to do that: you come from first year, where you wrote a few thousand lines of Java and were told what a variable, a conditional and a loop are. What you need now is something else, smaller and quicker: to carry that knowledge over to a new language.

And you will need to do it a few more times over the course of your degree. So let's stop for a moment to think about how it is done, instead of doing it in fits and starts, searching the internet for every line that won't come.

4.1 Where C comes from

In 1969, when Ken Thompson wrote the first version of Unix in PDP-7 assembly language —the story is in Chapter 1—, every operating system in the world was written in assembly. It was not a choice: it was the only way to do it. An operating system touches registers, interrupts and physical addresses, and no high-level language of the time let you get at them.

Thompson wrote a language, B, so as not to have to start from scratch on every machine. It was a simplification of Cambridge's BCPL and had a serious problem: it did not distinguish data types. Everything was a machine word. On a PDP-11, which could address bytes, that did not hold up.

Between 1971 and 1973 Dennis Ritchie kept adding types to B until it stopped being B and came to be called C [2]. And in 1973 something happened that nobody had done before: Thompson and Ritchie rewrote the Unix kernel in C.

C is not a language that was later used to write operating systems. It is a language that was invented *to* write an operating system, and all its decisions —the good ones and the bad ones— make sense from there.

Portability came out of that rewrite, and it is what made Unix leave Bell Labs and end up everywhere. The language’s oddities come out of it too: that it lets you add 1 to an address, that it doesn’t check whether you run off the end of an array, that it has no character strings. A language that has to be able to write a particular byte at a particular memory location cannot protect you much from yourselves.

The reference book is still Kernighan and Ritchie’s [1]: some two hundred and eighty pages that include the manual for the whole language. There are thousand-page C books; don’t trust them.

4.2 Six questions

When you face a new imperative language —and C, Java, Python, Go and Rust all are— there are six questions which, once answered, let you write in it any program you already knew how to write:

1. Which primitive data types does it support?
2. Which structured data types can I define in it?
3. How do I write sequential statements?
4. How do I write a conditional?
5. How do I write a loop?
6. How do I compile it and build an executable?

The first five are the language. The sixth is not the language but the environment; without it, though, you will never get to run anything, and it is where most time is lost on the first day. The rest of this chapter is six sections, one per question.

That the recipe works can be checked: Appendix B answers these same six questions for Rust, which is the language Linux has started to accept for its drivers. It isn't needed for this course; it is there in case it comes your way one day.

Careful

These six questions are the minimum to *write*, not to write *well*. From a language you also have to end up learning how code is organised into modules, what is idiomatic and what isn't, and how errors are handled. But that is learnt by reading other people's code, and to read it you first have to be able to decipher it.

4.3 Which primitive types?

Few, and all of them numeric.

There are the integers —`char`, `short`, `int`, `long` and `long long`—, which you can qualify as `signed` or `unsigned`, and the reals —`float`, `double` and `long double`. And that's it.

Three absences that will surprise you, coming from Java:

- **There is no boolean type.** In conditions, zero means false and anything else means true. Since C99 there is `bool` if you include `stdbool.h`, but underneath it is still an integer, and the code you will read doesn't use it.
- **There are no character strings.** The type `string` does not exist. A string is an array of `char` ended by a terminating null byte, and everything you want to do with it you will do with library functions (`strlen`, `strcpy`).
- **`char` is an integer.** A byte, to be precise. You can add to it, subtract from it and compare it, and that is exactly what we will do in this chapter's `c-count` to count newlines.

The absence that will surprise you most, though, is not a type but a size: **the standard does not say how many bytes an int takes**. It only says which values each one must be able to hold and that none can be smaller than the previous one. The real size is chosen by the compiler of the machine you compile on, which is why a program that assumes an `int` takes four bytes will break the day you change machines.

The `sizeof` operator gives you the size in bytes of any type or variable, and it is the right way to ask that question:

Listing 4.1. Sizes, asked of the one who knows

```

1 static void size(const char *name, size_t bytes)
2 {
3     char buf[64];
4
5     sprintf(buf, "%-12s %2zu bytes\n", name, bytes);
6     write(1, buf, strlen(buf));
7 }
8
9 int main(void)
10 {
11     size("char", sizeof(char));           ❶
12     size("short", sizeof(short));
13     size("int", sizeof(int));
14     size("long", sizeof(long));
15     size("long long", sizeof(long long));
16     size("float", sizeof(float));
17     size("double", sizeof(double));
18     size("void *", sizeof(void *));       ❷
19     size("size_t", sizeof(size_t));
20     return 0;
21 }
```

- ❶ This line always prints 1: the standard *does* fix `sizeof(char)` at 1. The byte is the unit of measure, and all the other types are measured in chars.
- ❷ A pointer is a type too, and it has a size too. On a 64-bit machine it takes 8, and that is, literally, what “64-bit” means.

```
alumne@labS0:~/S2$ ./c-types
char          1 bytes
short         2 bytes
int           4 bytes
long          8 bytes
long long     8 bytes
float         4 bytes
double        8 bytes
void *        8 bytes
size_t        8 bytes
```

The last one on the list, `size_t`, is not a primitive type: it is the name the library gives to the unsigned integer type large enough to measure any object on the machine. You will find it in half the system headers, and it will turn up again in the next question.

4.4 Which structured types?

Four constructors, which can be combined without limit.

4.4.1 The array

```
int  v[10];           /* ten integers, v[0] to v[9] */
char name[32];        /* thirty-two characters    */
```

The size must be known at compile time*, and the indices go from 0 to $n - 1$, as in Java.

Careful

What is **not** as in Java: nobody checks that the index fits. If you write to `v[10]`, C lets you: you will write over whatever bytes come after the array, whoever they belong to. The program won't

*Not always, since C99: there are variable-length arrays. We won't use them.

crash at that moment; it will crash later and somewhere else. Half the hours you spend debugging in this course will come from here.

4.4.2 The pointer

A variable that holds a memory address. It is declared with an asterisk and has two operators:

```
int a = 4;
int *p;           /* p is a pointer to int      */

p = &a;           /* & : the address of a          */
*p = 7;           /* * : what is at address p      */
                  /* now a is 7                    */
```

The pointer is the piece that makes everything else possible. A C program never passes a large structure to a function: it passes its address. The system call `read()` does not return the bytes it read: you give it the address where it has to leave them. Spend some time on it now, because from Chapter 5 onwards we won't be getting away from it.

Array and pointer are closely related: the name of an array, used inside an expression, is the address of its first element. That is why a function that takes a string takes it as a `char *` and not as a `char[]`, and that is why, inside that function, there is no longer any way of knowing how many elements it had.

4.4.3 The structure

A handful of variables under a collective name, like the fields of a class without methods:

```
struct point {
    int x;
```

```
    int y;
};

struct point p;
p.x = 3;           /* with a pointer it would be q->x = 3 */
```

There is a variant, the **union**, where all the fields share the same memory and only one is valid at a time. You will see it in kernel structures, where saving memory matters.

4.4.4 The new name

typedef doesn't create any type: it gives a new name to one that already exists.

```
typedef unsigned long size_t;
typedef struct point Point;
```

It saves you writing **struct point** fifty times and, more importantly, it gives names that say what a thing is instead of how it is made. When the manual says that **read()** returns an **ssize_t**, what it is telling you is “an integer big enough to count bytes, and signed because -1 means error”; how many bytes it takes is the machine's business.

That leaves **enum**, which is a list of named integer constants. C will not stop you from assigning an **enum** variable a value that is not on the list: remember where we come from.

4.5 How do I write sequential statements?

A statement is an expression followed by a semicolon. A group of statements is a block, between braces. So far, as in Java.

The surprise is that in C *everything* is an expression, and so everything has a value. Assignment has one:

```
b = (a = 3) + 1;    /* a is 3, b is 4 */
n = m = 0;         /* the same as m = 0; n = m; */
```

It looks like a curiosity and it isn't: this is where the most common idiom of systems C comes from, which you will see in the section on loops.

4.5.1 Functions

A function is declared with the type it returns, its name and its parameters:

Function `error_and_exit`

```
void error_and_exit(const char *msg, int exit_status);
```

This is a *prototype*: it says how the function is called, not what it does. It must be visible *before* the first call, which is why prototypes live in `.h` files and the code in `.c` files. If you don't do this, the compiler cannot check the call for you and will tell you so with an *implicit declaration* warning.

There is one function with a fixed name, `main`, which is where everything starts:

Function `main`

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

The shell of Chapter 3 is the one that fills in these two parameters: `argc` is how many words the command line had and `argv` is the array with those words, counting the program's name in `argv[0]`. The integer `main` returns is the program's exit status, which the system

call `wait()` will deliver to the parent (Chapter 5); by convention, 0 means “all is well”.

4.5.2 Writing to the screen

In this course we will always write with the system call `write()`, never with `printf` or with `cout`.

It is not to make your life hard. `printf` is a C library function that will end up calling `write()` when it sees fit; in between there is a buffer that decides on its own when to really write. That buffer is precisely what will hide the real order of things from you the day you have two processes writing at the same time, which will be soon. With `write()` you see the system, which is what this book is about.

The work is split into two steps. `sprintf` formats the text and leaves it in a `char` array; `write()` sends that array to a file descriptor:

```
char buf[128];

sprintf(buf, "Example number %d\n", 1);
write(1, buf, strlen(buf));
```

Look at the colour: `write()` comes out in red and `sprintf` doesn't. That is the rule throughout the book —system calls in red, library functions in black—, and here you can see clearly the difference between the two. `sprintf` doesn't leave your process. `write()` does: it asks the kernel to do the writing.

This pair will turn up in every example in the book, and it will turn up like this, in full, as in the lab: `write()` is precisely what you want to see.

What we do have packaged is the other thing that is repeated everywhere. A system call can fail, and when it fails it returns `-1` and leaves the reason in `errno`; `perror` turns it into a message. In the examples, when a call fails there is nothing else to do but say so and stop, and that is why they all use the same function from the lab, which lives in `tools.h`:

Listing 4.2. The one house tool

```

1 void error_and_exit(const char *msg, int exit_status)
2 {
3     perror(msg);
4     exit(exit_status);
5 }

```

The second parameter is the exit status the parent will receive; in the examples it is always 1, which means “something went wrong”.

And here comes the third colour, the last one. The lab’s function and `perror` are in **blue**, because they are neither one thing nor the other. The first is *ours* —you will find it in `tools.h` and nowhere else—, `perror` belongs to the library, and both do the same job: say what failed. That they are not red means what it seems to mean: what crosses the boundary is the `write()` that `perror` does underneath, not them.

With that we can now write the first complete program. It is the skeleton of all the ones you will write in the lab: check the arguments, do the work and return a status.

Listing 4.3. The skeleton of any program

```

1 static void Usage(const char *me)                                ❶
2 {
3     char buf[128];
4
5     sprintf(buf, "Usage: %s name [name...]\n"
6             "Greets each of the names it is given.\n", ←
7     me);                                                         ❷
8     write(2, buf, strlen(buf));
9     exit(1);
10 }
11 int main(int argc, char *argv[])                                ❸
12 {
13     char buf[128];
14     int i;

```

```
15
16     if (argc < 2)                                ❹
17         Usage(argv[0]);
18
19     for (i = 1; i < argc; i++) {
20         sprintf(buf, "Hello, %s!\n", argv[i]);
21         write(1, buf, strlen(buf));
22     }
23     return 0;                                     ❺
24 }
```

- ❶ The `Usage` function is a lab convention: when the arguments are not the right ones, it says how the program should be called, explains what it does and stops.
- ❷ Error messages go to file descriptor 2, not 1. That way, if someone redirects the output to a file —Chapter 3—, the error can still be seen on the screen.
- ❸ The usual signature. The names `argc` and `argv` are up to you; the types are not.
- ❹ `argc` is 1 when you haven't typed any argument, because the program's name counts.
- ❺ Status 0: all is well. `Usage`, on the other hand, ends with status 1.

```
alumni@labS0:~/S2$ ./c-hello Jordi world
Hello, Jordi!
Hello, world!
alumni@labS0:~/S2$ ./c-hello
Usage: ./c-hello name [name...]
Greets each of the names it is given.
alumni@labS0:~/S2$ echo $?
1
```

4.6 How do I write a conditional?

```
if (a > b)
    max = a;
else
    max = b;
```

No surprises, except one: the condition is not a boolean, it is an integer, and it counts as true if it is not zero. So this other thing is legal, compiles and doesn't do what it seems to:

```
if (a = b)      /* assigns b to a, and checks whether the result ←
    is zero */
```

i Careful

The classic = where you meant == is, by far, the mistake that will waste most of your time. Always compile with -Wall: gcc warns you about this particular case.

There is also **switch**, to choose among constant integer values, with two pitfalls: cases **fall through** from one to the next if you don't put a **break**, and **default** is optional.

And there is the ternary operator, which is a conditional that has a value and can therefore be put inside an expression:

```
max = (a > b) ? a : b;
```

4.7 How do I write a loop?

The usual three:

```
while (condition) { ... }      /* checks, and may never enter */
do { ... } while (condition);  /* enters, then checks          */
for (i = 0; i < n; i++) { ... } /* initialise; check; advance */
```

C's `for` is not modern Java's: it doesn't iterate over collections, there is no `for each`. The three parts are arbitrary expressions and any of the three can be empty; `for (;;)` is an infinite loop. Inside all three, `break` gets you out and `continue` jumps to the next iteration.

And now, the idiom we announced in the section on statements:

```
while ((nread = read(0, buf, SIZE)) > 0) { ... }
```

It is read from the inside out: it calls `read()`, stores the result in `nread` and, since assignment is an expression with a value, compares *that same* value with zero. In one line there is the read, the saving and the exit condition. You will find this form in all the systems code you read, and you had better recognise it now than stumble on it along the way.

This is the whole program. It counts the lines and bytes of standard input, and it opens no file: it reads from file descriptor 0, whatever that is.

Listing 4.4. A pocket-sized `wc -l`

```
1  #define SIZE 512
2
3  int main(void)
4  {
5      char buf[SIZE], output[64];
6      ssize_t nread;
7      long bytes = 0, lines = 0;
8      int i;
9
10     while ((nread = read(0, buf, SIZE)) > 0) {
11         bytes += nread;
12         for (i = 0; i < nread; i++)
13             if (buf[i] == '\n')
14                 lines++;
15     }
16     if (nread < 0)
17         error_and_exit("read", 1);
```

```

18
19     sprintf(output, "%ld lines, %ld bytes\n", lines, bytes);
20     write(1, output, strlen(output));
21     return 0;
22 }

```

- ❶ `ssize_t` and not `int`: `read()` returns how many bytes it has read, and `-1` if there has been an error. It needs a sign.
- ❷ The idiom. The loop ends when `read()` returns 0, which means there is no more data.
- ❸ Here you can see that `char` is an integer: we compare it with the character constant `'\n'`, which is 10.
- ❹ Leaving the loop doesn't mean everything went well. If `read()` returned `-1`, we have to say so.

And since the program doesn't know where its data comes from, the shell can decide that for it, with the redirections of Chapter 3 or with one of the pipes of Chapter 10:

```

alumni@labSO:~/S2$ ./c-count < /etc/passwd
82 lines, 4970 bytes
alumni@labSO:~/S2$ ls /bin | ./c-count
6808 lines, 77694 bytes

```

4.8 How do I compile and build an executable?

When you type `gcc -o prog prog.c` it looks as if a single thing happens. Four things happen, and it pays to know which one has failed you:

1. The **preprocessor** works on the text. It copies the files of the `#includes` where they are, substitutes the `#defines` and decides which pieces of code survive the `#ifdefs`. It doesn't understand C: it moves text around.
2. The **compiler** translates the result into the machine's assembly

language. It is the one that complains about syntax and type errors.

3. The **assembler** turns that into machine code: an object file, `.o`, with the list of names it provides and of the ones it is missing.
4. The **linker** puts the object files and the libraries together, resolves the pending names and writes the executable. If it says *undefined reference*, a function that nobody has implemented has made it this far.

The five `gcc` options you will be asked for in the lab are `-c`, `-o`, `-I`, `-L` and `-l`:

```
# only as far as the .o: preprocess, compile and assemble
gcc -Wall -c prog.c

# link the objects and write the executable "prog"
gcc -o prog prog.o tools.o

# -I: where to look for .h files
gcc -Wall -c -I../include prog.c

# -L: where to look for libraries. -l: which library
gcc -o prog prog.o -L/usr/local/lib -lm
```

`-l` is the surprising one: `-lm` means “the library `m`”, which on disk is called `libm.so` or `libm.a`. The linker adds the `lib` prefix and the extension.

4.8.1 Why split `.h` and `.c`

Because the compiler processes one file at a time and knows nothing about the others. From your module, the rest of the program needs to know the prototypes —how the functions are called— but not the bodies. The prototypes go in the `.h`, which everybody who uses them includes; the bodies go in the `.c`, which is compiled only once and produces a `.o` that the linker attaches to whoever needs it.

A `.h` always carries the guard you have seen in `tools.h`, so that including it twice doesn't cause an error:

```
#ifndef TOOLS_H
#define TOOLS_H
    ... prototypes ...
#endif
```

If you end up with a few `.o` files that always go together, the `ar` command packs them into a library (`libtools.a`) that you can then pass to `gcc` with `-L` and `-l`.

And before you hand anything in, run `indent` on it: it indents the whole file following a uniform style. Badly indented code is not an aesthetic problem; it is code where the missing brace can't be seen.

4.9 The makefile

You now have all the pieces, and you can already see the problem: every time you change a line you would have to remember which `.o` files have to be rebuilt and in what order. With three files it is tedious; with thirty it is impossible.

The `make` command does it for you, from a text file called `makefile` that contains **rules**. A rule has three parts:

```
target: dependency1 dependency2 ... dependencyN
[TAB] how to build the target from the dependencies
```

i Careful

That [TAB] is a real tab character, not eight spaces. It is the most frequent mistake in everybody's first makefile, and **make** reports it with a message that doesn't help at all: *missing separator*.

The minimal rule says that **c-hello** depends on **c-hello.c** and that it is built with a **gcc**:

```
c-hello: c-hello.c
    gcc -o c-hello c-hello.c
```

What **make** does with this is a single thing, and it needs saying properly: it looks at the dates of the target and of the dependencies and **only runs the command if some dependency is newer than the target**. If you haven't touched anything, it will tell you there is nothing to do. Everything else follows from this: dependencies are not decoration, they are what allows it not to recompile what is already fine. And if you get them wrong **make** will not warn you: it will simply leave you an old executable with the new code, and you will spend the afternoon on it.

Rules chain together. If a target depends on a file which is, in turn, the target of another rule, **make** goes down to that one first. With that, and with two variables so as not to repeat the compiler's name and the options, you already have the makefile for the lab session:

Listing 4.5. A complete makefile, and no more

```
# the bare minimum to build the c-count program of chapter 4
CC      = gcc
CFLAGS = -Wall -Wextra -g

c-count: c-count.o tools.o
    $(CC) -o c-count c-count.o tools.o
```

```
c-count.o: c-count.c tools.h
    $(CC) $(CFLAGS) -c c-count.c

tools.o: tools.c tools.h
    $(CC) $(CFLAGS) -c tools.c

clean:
    rm -f c-count *.o
```

Three more things and we will leave it:

- **The first target in the file is the one made by default.** `make` with no arguments builds `c-count`; `make tools.o` builds only that.
- **A variable** is defined with `CC = gcc` and used with `$(CC)`. It lets you change the options in a single place.
- **There are targets that are not files.** `clean` doesn't produce any file called `clean`: it is a command with a name. Since the file never exists, it is always considered out of date and always runs.

Question 1

If you type `make` twice in a row without touching any file, the second time it does nothing. Now run `touch tools.h` and try again. What gets recompiled, and why?

Answer

It rebuilds `c-count.o` and `tools.o`, because both rules have `tools.h` as a dependency and it is now newer than the `.o` files. And once rebuilt, the `.o` files are newer than `c-count`, so it links again too. `make` hasn't looked at the contents of any file: only at dates.

4.10 Exercises

1. Compile and run the `c-types` of Listing 4.1 on your machine and in the lab. Is the output the same? Is there any line that must come out the same everywhere?
2. Add to `c-types` a line for `sizeof(struct point)`, with this chapter's two-integer structure. Do you get the sum of the sizes of the fields? If not, look up what *alignment* means.
3. Modify `c-count` so that it counts words too. We will say that a word ends at a space, a newline or a tab. Compare the result with that of `wc`.
4. Remove `-Wall` from the makefile and change the `==` in the newline line to an `=`. Does it compile? What does the program count now? Put `-Wall` back and see what the compiler tells you.
5. In the example makefile, delete `tools.h` from the dependencies of `c-count.o`. Now add a parameter to `error_and_exit`, in both the `.h` and the `.c`. Run `make`. What happens, and why is it worse than a compilation error?

Bibliography

- [1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988. ISBN: 0-13-110362-8.
- [2] Dennis M. Ritchie. “The Development of the C Language”. In: *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. ACM, 1993, pp. 201–208. DOI: [10.1145/154766.155580](https://doi.org/10.1145/154766.155580).