Easy manual for C

Francisco Rodríguez Melgar

Computer engineer

December 12, 2022

«Since it is more what you ignore than what you know, do not speak too much.»

—Raimundo Lulio, scholar and saint from the island of Mallorca, Spain

Contents

List of Figures

List of Tables

List of Programs

1.   Introduction

In this document I hope to be able to explain, at least, the basic fundamentals of the C programming language. Also, I hope to offer reasons to learn it and I will try to communicate to the reader part of the beauty I find in it. Under this first header I will explain what is programming, which kinds of languages do exist and offer an explanation about how we will structure this document.

1.1.   What is programming?

Since this is a not very advanced manual, it is possible you have never had any experience programming. If this is the case, I will explain shallowly what "programming" is. Programming is, according to the Oxford dictionary: "the process of writing and testing computer programs". I am a simple guy and have not put a foot in Oxford University but, perhaps naively, I expected a more enlightening definition. It allows us to start, though, programming is to write programs, therefore, to know what programming is, we need to know what a program is.

A program is the set of instructions that a computer follows to perform a concrete task. As an example, if you were a computer, and we made a program to make you buy a coffee in a vending machine, the program that you as living computer would follow would be something like this:

1. Get up, if you're sit

2. Walk to the coffe machine

3. Choose the coffe you'd like to have

4. Read the price of the coffee

5. Put coins up to the price in the slot for coins

6. Push the button of the desired variety of coffee

7. Wait until it's done

8. Pick up the coffee, and be careful no to scorch yourself!

Put that way, it would be wonderful to tell to your computer, or to any computer, something like "solve this differential equation" or "predict the weather of tomorrow." Sadly, this is where the craftiness of the programmer comes in. Computers do not understand the language of humans. They do not know what weather is nor what a coffe is. Computers only understand mathematical operations (and not a lot) and logic operations. If you do not know what logic is, as a science, do not worry, we will talk about it later. The programmer must be able to turn a complex task into a set of instructions a computer understands.

Finally, we could say that programming is "articulate complex tasks in human language in terms of simple tasks that a computer understands".

1.2.   How does one program?

Now we know **what** programming is, let's see how it is done, in general terms. Following the methaphor I used before, a "program", as we understand it, is a text file (or several) where those instruction the computer needs to do something are. As I said before, computers understand a somewhat small number of instructions, and, as a matter of fact, they only understand binary code. A computer stores in its memory (what is commonly known as RAM memory) the instructions that it must execute. That memory only stores bits, digits in a numeric system that contains only two figures: zero and one.

If we apply the definition of the last chapter, to program, we must write programs in zeroes and ones. To understand the magnitude of this, the program Firefox, the web browser, occupies around 500 KB, or, what is the same, half a million of bytes. A byte is eight bits. This means that the programmer that, supposedly, wrote Firefox would have had to write a continous file of four millions of zeroes and ones. It is only logical to think this is not the case.

Since the earliest times of computer science and software development people have come up with **formal languages** that explain in a comprehensible way to the human being how a program must be, but that allow us to make a program composed of those zeroes and ones. This is where the different languages you may have heard of come in: C, C++, C#, Java, Rust... All those languages differ in that they're different ways (each one with its pros and cons) to compose a program that, after a process, the computer is able to understand. This process is **compilation**. To compile a program is turning it from that language humans can understand (and that you are going to learn to write, I hope with my help) into a pure computers' language. The code written in those languages is called **source code** because it is the source from which we will obtain (compile) our programs. In general, I am not going to make a distinction between the program (compiled program) and the source code. We will tell them apart by the conext.

So, if we add this information to what we had before, we could say that programming is: "to write a file in a programming language that can be compiled into a program the computer can execute directly".

2.   Environment set up

Maybe you're already impatient, or perhaps you stopped reading a long time ago, But I think that introduction was needed, at least for those that do not know what programming is at the most basic level. Now we are going to talk about how to prepare an **environment** to program. The environment is the set of tools we are going to use to write and compile our programs. The problem of C is that it's a language very "close to the computer", what does this mean? That it is more difficult to understand and write for people, therefore the preparation you will need to do to program in C is a little bit more complex that if you used other languages. So let's go little by little.

2.1.   Operating system

If you're reading this manual, or this part, I understand you didn't explore programming before. Let's start by the beginning. Since C is a language computers understand more easily, we must know which operating system we have. If you haven't altered your computer in any way, the most probable thing is that you have a computer with Windows. Ideally you should install Linux, or create a virtual machine with it, or use the Windows subsystem for Linux.

Since explaining all the alternatives would make this manual very long and also would force me to make distinctions in each one of the following sections, I am going to suppose you are using the Windows subsystem for Linux, or WSL. The first step is to install the Windows characteristic that allows us to do that. Hit the Windows key and the letter R. Write in the little window that pops up `optionalfeatures` and hit enter. A window will pop up where you should look for the element "Windows subsystem for Linux", check the box in the left of the option and push accept. Restart when it is asked for it. After it, you will go to the Microsoft Store app and will look for "Ubuntu". Go to the first result and install it. After that, go to the start menu and open the app (Ubuntu). It will take a moment to install. After it completes, it will ask you to input an username and a password. Just a note: when you start writing the password you will not see anything, don't worry, it's supposed to be like that, just write the password, it will ask you to input it twice, if you did it differently, it will tell you to do it again. Be sure to remember the username and password. When you are done, you will looking at a black screen with a text that will read `{user}@{machine_name}:~$`. Congratulations, you have installed a Linux you can use in Windows.

This black window that only contains letters is called a terminal, and it is a way to interact with the computer that has been in use since decades ago. Instead of clicking icons, you will write commands in the terminal and you will hit enter. I am going to give you a series of basic commands and concepts so you can use it. In a terminal, at any given moment you are in a **work directory**, for example, if you write `pwd` and you hit enter, it will tell you in which directory you are in. Directory is just a fancy word computer people use to say what we call folder when using computers, a place where other files and folders may be put. I will show you an example of how it would look.

```
john@DESKTOP-U8OA808:~$ pwd
/home/john
```

With the command `cd` you move your work directory. Every directory has two special directories inside, the dot directory (`.`) and dot-dot (`..`). The first one is the same directory, that is: if you perform `cd .` you will stay in the same directory. The second is the parent directory, for example:

```
john@DESKTOP-U8OA808:~$ pwd
/home/john
john@DESKTOP-U8OA808:~$ cd ..
john@DESKTOP-U8OA808:/home$ pwd
/home
john@DESKTOP-U8OA808:/home$
```

In order to give you a better idea of what a filesystem is, I am going to explain it to you more thoroughly. In Windows, all your files and directories are in drives, which have letters assigned and end with a colon (`:`). For example, the most common path for the desktop is `C:\users\userName\Desktop` (userName being the name of the user whose dekstop we're talking about, let's say John). On the contrary, in Linux this is not this way, all directories come from the root directory (`/`). Note: if you haven't noticed yet, in Windows directory paths are written with backslashes (`\`) and in Linux with forward slashes (`/`). Each drive in Windows tends to be a physical drive: a memory stick, a hard disk, an SSD... In Linux, when you insert a drive or disk, it will simply **mount** in a directory like any other. That is: the files and directories of the new disk will be put in one point of the directory tree we already had.

These directories work like a series of dots joined by connections. Each directory or file is a dot, connected to the directory it hangs from, and having all the directories it has inside hanging from it. This may be drawn like I'm going to show you now:
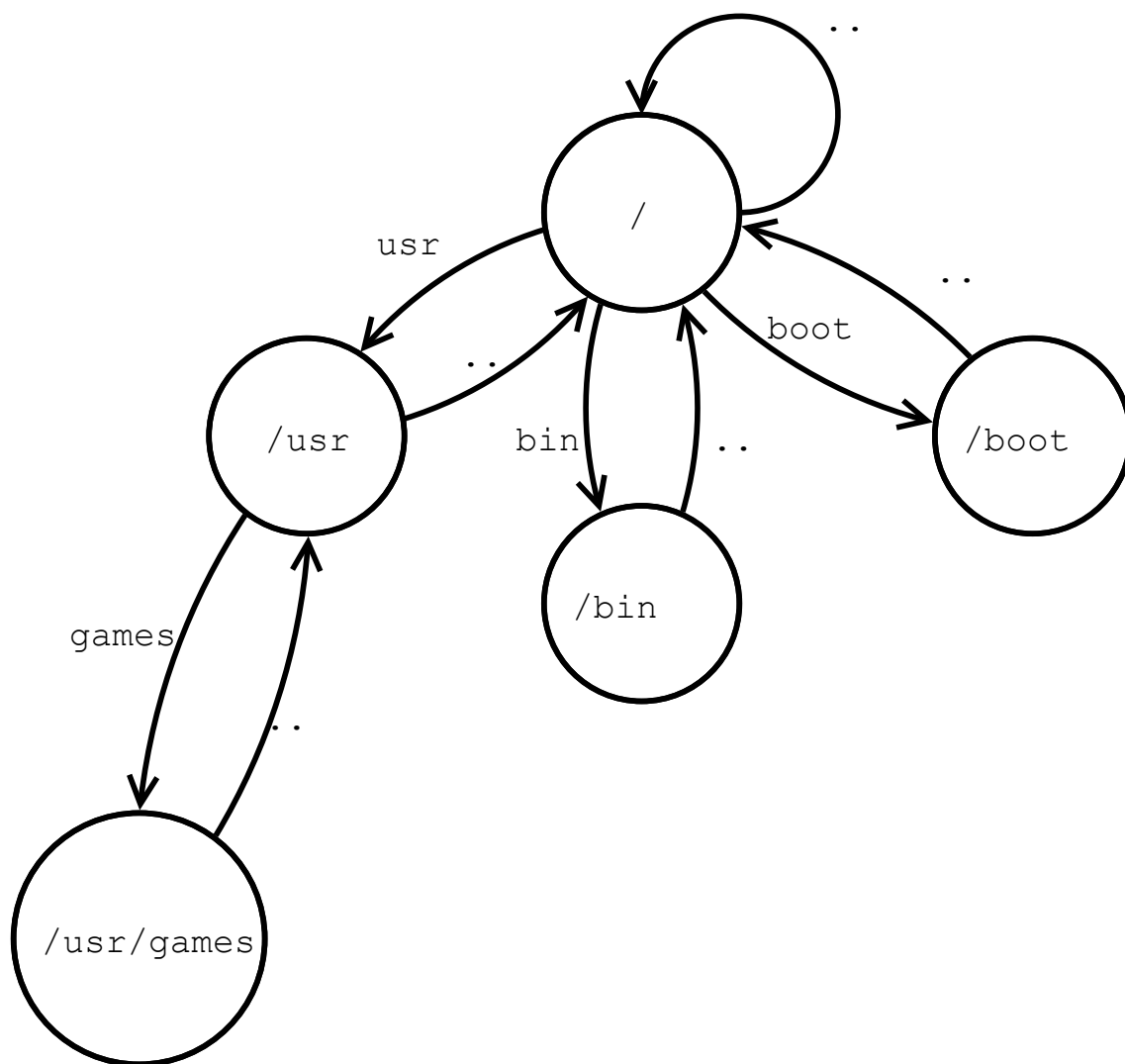
Figure 1: Example of a filesystem

In this figure, each directory has linked to it those inside it, and all of them show where the directory `..` which is inside them goes. As you can see, to reach a path like `/usr/games` we only have to "hop" from a directory to the next. If we wanted to come back, we can use the fictional directories that point to the father of the directory we're in (`..`).

The case of the root directory is special, because its parent directory is the same directory. In Windows the system is similar, but there is not only one root, there are several. Those being the different storage drives we have. Also, we do not need to be all the time hoping in one level jumps, we can use complete paths to navigate from a place to the other. If you haven't figured it out yet by the context, a path is simply the succession of directories that go from one to another. There are two types of paths:

1. Absolute paths: Those are the ones that start by slash, and they indicate a path from the root directory to a concrete directory or file. For example: `/home/john/music/Beethoven_symphony.mp4` would be and absolute path.

2. Relative paths: they are those that do not start from the root directory, but from the work directory. For example: `music/Beethoven_symphony.mp4` is relative path that will point to an existing file only if inside the current work directory there is a directory called `music` and, inside it, a file named `Beethoven_symphony.mp4`.

The text it shows up everytime you hit enter is called a prompt and, in general, it shows you your username, the machine where you are and after that, your working directory, if it fits in the screen. I am going to substitute the prompt just for a dollar sign in the examples I show you, so it fits better in the page. When you first started your terminal it showed a tilde (`~`) because that's an alias of your home directory, which is a route in Linux systems where the user stores his personal files. Generally speaking, that route is in `/home/<username>` for example `/home/john`.

Now we are going to learn to see what is inside a directory, the command that allows us to do so is `ls`, if you type it and hit enter you'd see... nothing. That's because we have not created any file in our home directory, the command to create files is called `touch`. Write `touch test.txt` and hit enter, if you now perform `ls`, you would see it will show up.

```
$ touch test.txt
$ ls
test.txt
$
```

`ls` has a lot of options, options of Linux commands are set with a dash infront of them, so, if you're told "ls with the options a and l" you must write `ls -l -a` or, joining all them in the same dash: `ls -la` Something like this should appear on the screen.

```
$ ls -la
total 8
drwxr-xr-x 1 john john  512 Jul  8 19:05 .
drwxr-xr-x 1 root root  512 Jul  7 22:37 ..
-rw-r--r-- 1 john john  220 Jul  7 22:37 .bash_logout
-rw-r--r-- 1 john john 3771 Jul  7 22:37 .bashrc
drwxr-xr-x 1 john john  512 Jul  7 22:37 .landscape
-rw-rw-rw- 1 john john    0 Jul  8 18:25 .motd_shown
-rw-r--r-- 1 john john  807 Jul  7 22:37 .profile
-rw-rw-rw- 1 john john    0 Jul  8 19:05 test.txt
```

As you can see, there are many files you have not created. This is because the `a` option makes `ls` to show us **all the files**, including the hidden ones, which are hidden because their name starts with a dot. The option `l` makes the command to show the files in a list, with more information about them. If this is a bit intimidating to you, it is normal, and I have good news. WSL sees the directories you have in you Windows system, so you can create a folder in your desktop and work with the Windows explorer, to create or delete files.

2.2. Installing the compiler

Now we already have a Linux installed in the computer, let's install the compiler. To do that, we are going to execute the commands the are shown next. If while executing any of them you are asked if you want to go ahead, answer yes.

```
$ sudo apt update
#Lots of text will be shown here, don't mind it.
$ sudo apt install build-essential
........
After this operation, 189 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
........
```

Now you should have the compiler installed, it is called GCC, it is an achronym meaning GNU Compiler Collection. To see if that is the case, type the following command. And check the output is similar to the one shown here.

```
$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.3.0-17
    ubuntu1~20.04' --with-bugurl=file:///usr/share/doc/gcc-9/README.
    Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,
    gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9
     --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker
    -build-id --libexecdir=/usr/lib --without-included-gettext --enable
    -threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu
    --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default
    -libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-
    verify --enable-plugin --enable-default-pie --with-system-zlib --
    with-target-system-zlib=auto --enable-objc-gc=auto --enable-
    multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --
    with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=
    generic --enable-offload-targets=nvptx-none=/build/gcc-9-HskZEa/gcc
    -9-9.3.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --enable-
    checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu
    --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
```

If you get a similar text, congratulations, you have installed the C compiler. Now we are going to, finally, start learning the concepts of the language.

3.   Your first program; say hello to the world!

Let's navigate into that folder you have in the desktop. Units of your Windows computer (the drives C, D, E...) are presented in the WSL as directories inside the path /mnt. So, C: will be under /mnt/c. I leave next an example on how to navigate to a folder called hello_world in your Windows desktop. You can create it as you're used to now.

```
$ cd /mnt/c/Users/John/Desktop/
$ cd hello_word
$ pwd
/mnt/c/Users/John/Desktop/hello_world
$
```

Now you are already in the folder, open it in the file explorer, because we have the last configuration left. In the Windows file explorer, in the top of the window, click on the View tab and check the box next to "show extensions of known file types". Next there is a picture of how it looks.
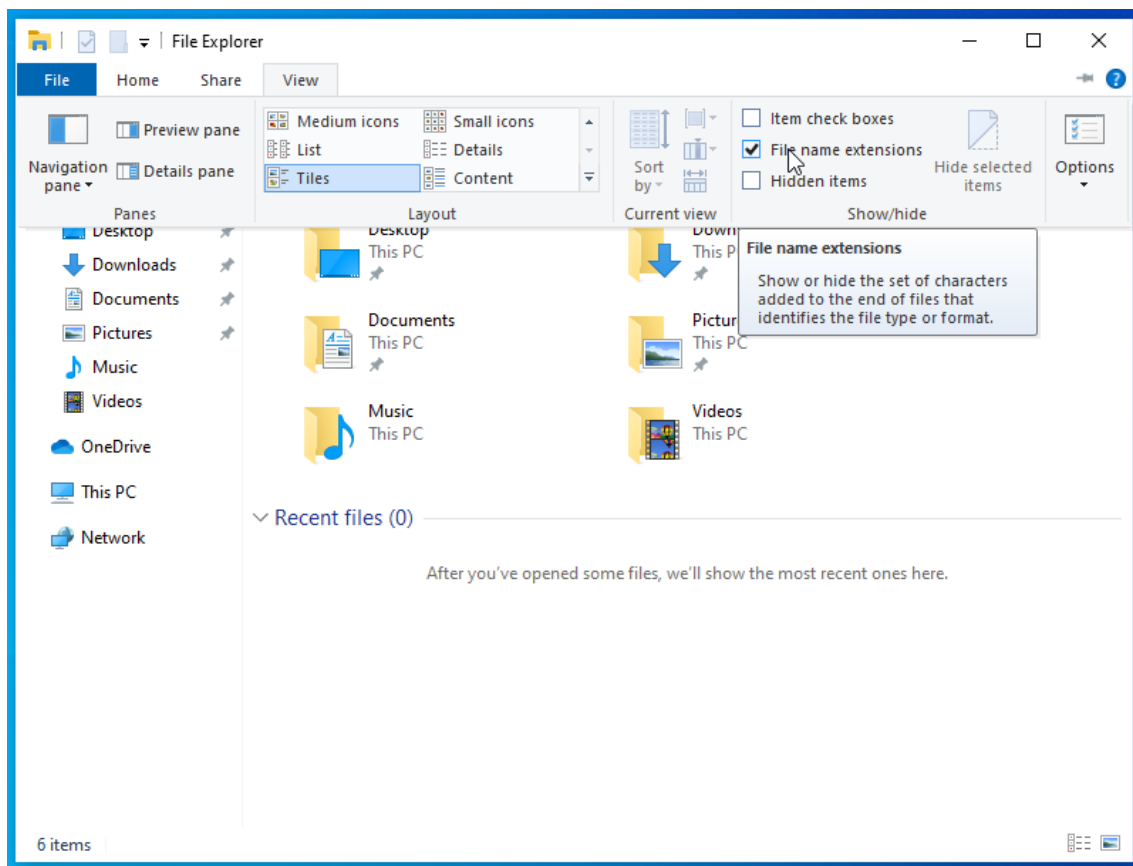
Figure 2: Configure how to see the extensions of known file types

Now, in the way you like the most (the terminal or the mouse), create a file in that folder called hello_world.c. Open it with the Windows Notepad, and I mean the Notepad, I do not mean WordPad, or Word. In that document we are going to write the following.

```
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
}
```

I want to clarify that the line that is displaced to the right is so because it has spaces in the left side. You can write two, four or eight, I recommend four as a general rule. Save the file, and go back to the terminal. You should have the file in the directory. Make `ls` to check it is in. Also, remember you need to navigate to the folder if you closed the terminal or changed work directory. Now we are going to compile our first program. To do that, we will invoke GCC. I will list the commands and their expected output next.

```
$ ls
hello_world.c
$ gcc -o hello_world.elf hello_world.c
```

When doing this, a new file will appear in the directory, called `hello_world.elf`. Congratulations! that is your first program in C. What does it do? It prints "Hello, world!". If you make double click on it, you will see that Windows does not know how to open it. That is because it is a Linux executable. Because of that, in the terminal, write `./hello_world.elf`. If you remember what I told you before, a path that does not start with the root directory is a relative path, and the directory dot `.` is a relative path. When the terminal gets a command from the user, it looks for a program with that name in a series of directories that are configured in your operating system. If you want to execute any other program (or other things you can execute, but I won't get tangled on them now) you must indicate a path, relative or absolute. To make the terminal understand we are introducing a path and not a command, we start it by dot, that is, this very directory. When you hit enter, you should see something like this:

```
$ ./hello_world.elf
Hello, world!
```

You you have compiled your first program, may be you are a bit disappointed, since you do not understand what it does, or how it does it. Hence it is the moment we adquire a compromise with each other. That is, if you do not understand something that appears in the programs we are going to see together, you will trust I will clarify eventually when the moment arrives, from my side, my compromise is that I will do it the earliest I can, so you have to put the least amount of effort in ignoring things you don't know.

For now, I will explain what the command we have issued before is, gcc is, as I said before, the C compiler, the option `o` (remember that I explain what options were when I explained you how to use `ls`) indicates that the next thing we are going to write is the name of the program we want to create and, finally, the name of our source code file. If you put `main.elf` the resulting file would be called that, or anything you wanted, I'll be compiling my programs as to files called `main.exe` mainly.

3.1. Text editor

You edited your first program with the notepad, but editing code in that way if a bit unbearable. Partially this is so because it is usual to use specialized editors that colorize the words that are important in the concrete programming language you are using and help you with things like knowing where the bracket that closes the one your looking at now. I am going to leave here a list of some common editors used with C code. I will not assume you are using any concrete one, so choose the one you like. I use the first one, but may be for your first steps you may stick to Notepad++, which is simpler, and jump to another one when you feel more comfortable writing more complex programs.

1. Visual Studio Code.

2. Atom.

3. Sublime text

4. Notepad++

4.   First steps

Now you already know what a source code file is I am going to show you how we are going to include code fragments in the manual. Let's revisit the program `hello_world.c`.

```c
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("Hello, world!\n");
5  }
```

Program 1: Hello World in C

As you can see, the lines are numbered, and there are words in blue, and others in red. The blue word are the ones called key words, for now, remember that all your programs must contain the lines 1, 2, 3 and 5 and that, between the lines 3 and and 5 you will write the instructions that will make up your source code files.

4.1. Variables

We are getting into business, at last! One of the first things that are needed in a program are **variables**, a variable is an abstraction to designate a space in the memory of the computer in which we store things. When we create a variable we say to the computer: "save this space in your memory to store data". C is a language of the kind we call typed, that is, each variable has a type and cannot change that type after it is created. In C there is a set of basic types that I will show you in a handy table.

Name	Size (in bytes)	Range	Ussage
char	1	$[-128, 127]$	One text character or a byte
short	2	$[-32\,768, 32\,767]$	Number in that range (generally network ports)
int	4	$[-2\,147\,483\,648, 2\,147\,483\,647]$	General type for integer numbers
float	4	$[\pm 3.4 \cdot 10^{-38}, \pm 3.4 \cdot 10^{38}]$	Simple precision decimal numbers
double	8	$[\pm 1.79 \cdot 10^{-308}, \pm 1.79 \cdot 10^{308}]$	Double precision decimal numbers

Table 1: Basic types of C

A table like that one may be intimidating at first, but it is simple. When we declare (create) a variable, we must say what type it has. I like to say that variables are like boxes and that, according to their type, inside that box some things fit and some others do not. To declare a variable, you write its type and a name, and do not forget to end the line with a semicolon (;)! Apart from declaring them, we must learn to give them a value. That is called "assign a value", and it is done with the equal sign (=). To give it a value you write the name of the variable, the equal sign and the value, let's see some examples and I'll get into some caveats.

Regarding the name: the name of a variable is made out of letters, numbers and underscores. The name of a variable must not be written all in uppercase, but it can contain some. It cannot start by a number, and **you should not start it with an underscore**. In general, you can use two notations to write variable names in C (and in any programming language):

1. **Camel case**: if a name contains several words, they must be written together, with the first letter of each word in uppercase, excluding the first. For example: `betterValue` or `targetNumber`. It is called like this because the uppercase letters remind of the humps of a camel.

2. **Snake Case**: The words are separated by underscores, all in lowercase, for example: `better_value` or `target_number`. It is called like that because the shape of the names written in this fashion remind of a snake that has eaten animals and has bulges in its body.

```c
1  #include <stdio.h>
2  int main(void)
3  {
4      char letter = 'a';
5      char byte = 120;
6      short shorty = 5520;
7      int money_i_want;
8      float money_i_have = 3.22F;
9      double money_you_have = 52.55;
10
11     money_i_want = 450000;
12     shorty = 11111;
13 }
```

Program 2: Declaration and assignment of variables

Computers do not understand about letters, only numbers, therefore, when you tell the computer that `letter` is equal to `'a'`, you tell it it's equal to the number that letter has assigned. Mind that to assign the value of a `char` you must use simple straight quotes. The correspondency between letters and numbers is written in the ASCII table, if you want to read it, I leave this link to a site where u can check it up. If you look there, you will see that the letter a has the value 97. After that, we assign to other char a numeric value, in line 7 you can see we declare a variable without assigning a value to it. That is totally okay, but beware!, **a variable to which you have not assigned a value has a random value**. This is why many teachers would advice you that everytime you declare a variable you should give it a value immediately. This process (giving value to a variable for the first time) is called "initialize" a variable.

In lines 11 and 12 we give value to variables that we have declared before, and I want to take some time talking about line 12. At first, it could seem that when writing `shorty = 5520;` we are enunciating a mathematical equality, that that is always going to be the same, but in C we do not work with "laws", but with instructions, so you shall not read that line as "the value of shorty is 5520" but "I have assigned the value 5520 to shorty". That is: you have put into the "box" a 5520, but nothing avoids you to pull that value out and put another one in as we do in line 12.

The values a programmer writes in the source code are called "literals". I think that the name is pretty much self explanatory. Each literal has a type, in the same way variables have. Later we will see why that's important. For now, remember that literal is the name of values the programmer writes explicitly in the code. That would be, numbers (like 5520) or letters, (like `'a'`). Apart from literals, we can assign to a variable the value of an **expression**, a expression is any text written in the C language that has a value. For now, we are going to limit ourselves to assign one variable to other variables.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 3;
5      int b = 2;
6
7      a = b;
8  }
```

Program 3: Assigning variables to other variables

In line 7 we can see how I assign to the variable a the value of the variable b, therefore, it will have a value of 2, now. Since seeing this is difficult, I am going to include in the next example a series of lines with the word printf in them, you may remember it from our first program. Later on, I will teach you to use it, but, at the moment, simply copy this program in your file of source code and compile it as we did before. (Copying things from a PDF tends to be problematic, therefore type it yourself, also, it will serve as typing practice.)

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 3;
5      int b = 10;
6      printf("a is equal to: %d\n", a);
7      a = b;
8      printf("b is equal to: %d and a is the same, that is: %d\n", b, a);
9      b = 22;
10     printf("b is equal to: %d, a still is %d\n", b, a);
11 }
```

Program 4: Final example of variable using

If you type this program, compile it and execute it, you should see something like this:

```
$ ./main.exe
a is equal to: 3
b is equal to: 10 and a is the same, that is: 10
b is equal to: 22, a still is 10
```

As you can see, when a = b is written, **the values of a and b are not linked together**. Nevertheless, when assigning any value to a variable, you must be careful. Going back to the methaphor of the boxes, in a box you can fit things of a certain set of shapes and sizes. If a data is, for instance, a decimal number (wether it is a float or a double) if it is assigned to an integer variable it will lose its decimal part. But there is more, if you apply this logic, what would happen if you assign a number such as 1203 to a char? If you go to the table 1: Basic types of C, you will see that 1203 is outside the range of char. What happens is... you do not know what happens. The elegant way of saying this is: "undefined behaviour". The next program is an example of that.

```c
1   #include <stdio.h>
2   int main(void)
3   {
4       char c = 1500;
5       short s = 5555555;
6       float f = 3.8e105;
7
8       printf("c: %d\n", c);
9       printf("s: %hd\n", s);
10      printf("f: %f\n", f);
11  }
```

<div align="center">Program 5: Erroneous assignments</div>

If you compile it, the compiler will throw a series of messages called warnings. Those warnings warn you that, while something is correct, it seems it contains some error. For example, if you wrote "I did not know your Belgian" the sentence would be technically correct, it would mean you didn't know a Belgian person related in some way to the person you're talking to, regardless, more probably, you wanted to say "I did not know you're Belgian ". If we compile and execute, something like the following should appear in the terminal.

```
$ gcc -o main.exe main.c
main.c: In function 'main':
main.c:86:14: warning: overflow in conversion from 'int' to 'char'
   changes value from '1500' to '-36' [-Woverflow]
   86 |     char c = 1500;
      |              ^~~~
main.c:87:15: warning: overflow in conversion from 'int' to 'short int'
    changes value from '5555555' to '-15005' [-Woverflow]
   87 |     short s = 5555555;
      |               ^~~~~~~
$ ./main.exe
c: -36
s: -15005
f: inf
```

As you can see, neither the `char` is equal to 1,500 nor the `short` is 5,555,555, because they cannot be. The compiler does not say anything about the `float` or the `double`, the reason is that both data types have a special value called infinity, which symbolizes infinity. This is because, as we will see later on, they do not represent numbers in a totally correct way, and that's why they can be positive and negative infinity.

Let's take some time for the problem of decimal numbers. If you know how binary code works, you would know that a binary number of n bits you can represent 2^n numbers. In the case of decimal numbers, we use one complex system called IEEE 754. I am not going to go into detail here, but the main problem of this way of representing numbers is that it is not only not exact (for example, the number 0.1 cannot be represented exactly), but its precission is not constant. What does this mean? That if near the zero the float may distinguish between 1.10 and 1.11, it is possible they cannot distinguish 10000000.10 from 10000000.11. Be careful about that when you use decimal numbers.

As you have seen here, there are allowed assignments and forbidden ones. The allowed ones (those the compiler does not see as something bad), are called implicit conversions, their names comes from the fact you do not need to do anything to make them happen, for example, assigning a char value to an integer variable. Later on I will teach you to perform conversions between data types explicitly.

4.2. Printing things

Programmers call "print" to write things into files and, specially, in the screen, like your first program that wrote "Hello, world!". I do not want to get ahead of myself, because there are several concepts behind what we use to print things on the screen, nevertheless, I need you to be able to show things on the screen to be able to test your own programs.

To print things on the screen you must use the word `printf`. With a syntax (syntax just means the way things are suppossed to be written) a little bit difficult. You must write `printf`, an opening parenthesis and a thing called "format" which is the text that is going to be printed, surrounded by double straight quotes (`"`). To include variables in that printing, you must put "specifiers", which are special texts that signal **the type** of the variables you want to print. After the format, we will write the variables we are going to print, separated by commas, in the order we wrote their specifiers. Also, there are some special characters you must write in a special manner: the new lines and the tabulators. This is a bit confusing, so I will show you two tables where you can see the specifiers and the special characters.

Specifier	Type it prints
%d	Integers (`int`)
%f	`float`
%lf	`double`
%hd	`short`
%c	`char` as characters (no numbers)
%s	Text, written as `"A text"`, quotes included.
%p	Pointers, they are an advanced feature of the language, I will explain them later

Table 2: Format specifiers

On the other hand, the special characters are these, and are written with a backwards slash (`\`) infront of them. In the table I already included the backward slash. Also, since the specifiers are written starting with percentage symbol (`%`), if you want to print it, you need to put it twice.

Sequence	Printed character
\\	Backwards slash
\n	New line
\t	Tabulator (prints spaces until the next character is aligned with four character column in the terminal)
%%	Will print just one percentage symbol.

Table 3: Sequences to print special characters

This is a bit dry, let's see an example.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int integer = 654654;
5      short shorty = 25254;
6      char charty = 'a';
7      double decimal = 2.3;
8      printf("The integer is:\t%d\nThe short is:\t%hd\nThe char is the
           letter:\t%c\nThe decimal number is:\t%f\n", integer, shorty,
           charty, decimal);
9  }
```

Program 6: Ejemplo de impresión.

The line is very long and in this page is it written as many, but you should write it as a single line. If you look closely, there is only one line number, that indicates that it's the same line but it is broken so it fits in the page. The program, once compiled and executed, should print something like this.

```
$ ./main.exe
The integer is: 654654
The short is:   25254
The char is the letter: a
The decimal number is:   2.300000
```

As a bottom line: `printf` does not add anything you do not put in it, like new lines, so if you want to print the next thing in a new line, remember to write \n at the end of the format. Also, it is good that your program prints a new line character as the last thing, if not, it could leave things unprinted. That's because the terminal forces itself to print everything you have ordered it to print when it finds a new line.

4.3. Operators

Playing a shell game with the values of the variables you declare in a program is boring, I know, therefore we are going to learn to perform operations with them. In C (and in any other programming language) there are the so called **operators**. They are symbols that allow us to perform calculations. Operators are a mathematical concept, and are applied to a set of arguments, or better put, operands. In math, the symbols $+$, $-$, $\times$ and $\div$ are mathematical operators for addition, substraction, multiplication and division, respectively. In the same way we have done with the basic types, I will present the operators in a table and later on we will see examples on how to use them.

Operator	Description
+	Addition. Adds integers and decimals together and between them.
-	Substraction. Substracts from the left operand the value of the right operand.
/	Division. Return the result of the division of the left operand by the right one. Note that, **if both operands are integers**, the operator performs integer operation, that is, **without decimals**.
*	Multiply. Asterisk has many functions in C, but this is the first of them you will discover. The type of the multiplication of two integer is always a **double**.
++	Increment. Makes a number (either integer or decimal) go up one unit, it can be prefix (before the operand) or postfix (after the operand).
--	Decrement. Works as the increment, but it makes the number to go down one unit.
%	Module. Is an operator that return the residue of the division of the left operand by the right operand.

Table 4: Basic math operators in C

In the last section I told you that we could assign a value to a variable. I also told you that an expression is a fragment of C code with a value, and that the name of a variable alone is an expression. Now we have operators, we can write more complex expressions, for example a+b would be an expression whose value would be the addition of a and b. We can make calculations now!

You must be careful, though, because, as I said before, all expression in C has a **type** and, as we saw in the last section, assigning a value of incorrect type to a variable is error prone. For example, the operator division behaves differently if the operands (the numbers we're dividing) are integer or decimal. Let's see and example.

```c
#include <stdio.h>
int main(void)
{
    double d1 = 1/3;
    printf("d1: %f\n", d1);

    double d2 = 1.0/3;
    printf("d2: %f\n", d2);
}
```

Program 7: Integer division vs decimal division

If you compile and execute the problem you will see this result:

```
$ ./main.exe
d1: 0.000000
d2: 0.333333
```

What would look like the same operation gave totally different results, and this is because the **type** of the operands was different. In C, a literal integer number value is an `int`, and the division operator when it is operating on two integers, has an integer type. Nevertheless, when **any of the two** operands is decimal, the operator performs the decimal division, and its type is `double`.

Operators `++` and `--` are special, because they are unary operators. An unary operator is an operator that is applied only to one operand. For example, in math you have the operator square root, whose symbol is $\sqrt{}$ which, applied to just one number, gives us the number (or numbers) that squared give us the operand. The operators increment and decrement are unary and, also, can be written infront or behind the operand. These are special operantors that do not only give us a value, but also **affect the value of the operand they act on**. Simply: if `a` is equal to three and we make `a++`, `a` will be four, but if we assign the value of the operation to other variable, that other variable will have a different value depending on if we write it in the postfix or prefix way, let's see it in the code.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 3;
5      int b = ++a;
6      printf("a: %d; b: %d\n", a, b);
7      a = 3;
8      int c = a++;
9      printf("a: %d; c: %d\n", a, c);
10 }
```

Program 8: Increment and decrement operators

If you execute it, you'll see the following.

```
$ ./main.exe
a: 4; b: 4
a: 4; c: 3
```

I want you to understand precisely what is happening here: everytime we apply the operand increment to a variable, that variable will increment its value in one unit. Nevertheless; depending on if we write it as prefix (infront the variable) or postfix (after it) the expression itself will have one value or another. If we do it prefix, the value of the expression of the operation would be the value of `a` **after** it increments its value, if we do it as a postfix operator, the value will be **without the increment**.

Following the line of increment and decrement operators, there are also operators that put together the assignment with other mathematical operations, that is, substituting for example `a = a * 3;` by `a *= 3;`. The same style of operators exists for substraction, addition, division and module.

Finally, we simply must tell that the **priority** of the operations is the same than in mathematics: the first expressions to be evaluated are those inside parenthesis, after them division and multiplication, both have the same priority, so in case you have several mixed, you will execute them from left to right. After that, addition and substraction that are, again, executed from left to right.

4.3.1. Casting: explicit conversions

It is usual that one needs to convert one type into another, for example, in the case we saw before with the division, if you wanted a decimal division, you would need to make one of them decimal to get the result you want. It is also common than a variable that is an `int` gets assigned to a `char` because you know its result is inside the range of values a `char` can hold. Casting is a word that means the process of putting molten metals inside a shaped container what will make the metal to retain that shape when it solidifies. Nevertheless, it has its limits, laws of logic still apply, and you cannot make a `char` to hold more than 127, for example, regardless of casting.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 3;
5      int b = 2;
6      double result = (double)a / b;
7  }
```

Program 9: Casting example

In line 6 we need one of the operands to be decimal (either `float` or `double`). We perform cast to that type on the first operand. The syntax is easy, but I am going to explain it in detail. You must write the name of the type you want to convert the value to between parenthesis and next to it the expression you want to cast. I'll leave you a couple examples:

1. `(int)(decimal_number / other_decimal)`: here we are casting to an integer type the result of a division, as you can see, we need to enclose the whole division in parenthesis so the casting is not applied only to the first element.

2. `(char)(number % 128)`: Here you can see one of the instances in which casting some integer type to a smaller one is ok. If `number` is an integer, by casting it to `char` there could be problems but since we have performed module on 128, the result of the expression is going to be between 0 and 127, therefore we know it is going to be in range of the `char`.

Following there is a table that will tell you which conversions are allowed and which are not possible.

		Destiny type				
		char	short	int	float	double
Source type	char	OK	OK	OK	OK	OK
	short	Casting (overflow)	OK	OK	OK	OK
	int	Casting (overflow)	Casting (overflow)	OK	OK (precission)	OK (precission)
	float	Casting (rounding, overflow)	Casting (rounding, overflow)	Casting (rounding, overflow)	OK	OK
	double	Casting (rounding, overflow)	Casting (rounding, overflow)	Casting (rounding, overflow)	Casting (rounding, overflow)	OK

Table 5: Type conversions in C

Where I write "ok" I mean that there is an implicit conversion, but you must be careful, the range of the integer is big and you may find that the precission of a `float` is not good enough in the bigger values to have problems distinguishing one unit from the next. I am going to be sincere with you, this does not happen, the `float` is precise enough in the limits of the integer to tell whole units apart, but I write it in the table to remind you that you may look at the precission of the integers. Where I write casting and I precise there can be an overflow I mean you're performing cast from a bigger type to a smaller one, so that is not generally acceptable unless you check yourself that the value of the casted expression fits into the new type.

4.3.2. Final example of a program with operators

At this point, we can make our first program that does "something", I am going to give you an example that calculates the solution to a system of linear equations, that is:

$$\begin{cases} ax + by = c \\ dx + ey = f \end{cases}$$

With a system like that, we can apply substitution:

$$(1)\ ax + by = c \rightarrow x = \frac{c - by}{a}$$

$$(2)\ dx + ey = f \rightarrow x = \frac{f - ey}{d}$$

$$(1)\ \text{and}\ (2) \rightarrow \frac{c - by}{a} = \frac{f - ey}{d} \rightarrow y = \frac{af - dc}{ae - db} \rightarrow x = \frac{f - e \cdot \frac{af - dc}{ae - db}}{d}$$

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int a = 1;
5       int b = 3;
6       int c = 8;
7       int d = 2;
8       int e = 7;
9       int f = 12;
10
11      double y = (a * f - d * c) / (a * e - d * b);
12      double x = (f - e * y) / (d);
13
14      printf("%dx+%dy=%d\n", a, b, c);
15      printf("%dx+%dy=%d\n", d, e, f);
16      printf("x = %f; y = %f\n", x, y);
17  }
```

Program 10: Linear equation system solving

If you copy this program, compile and execute it, you would see that I have explained about variables and expressions. Nevertheless, the program has a problem: it only solves one system of equations, to change the system, we need to change the source code and recompile. This is not practical, and real programs do not work in this way. For now most of our programs would be like this one, because I want to explain more fundamental things first. Up until now our programs have been very boring, they are limited to execute a series of instructions one after the other. In real life, though, programs execute one set of instructions or other depending on conditions, or they repeat certain instructions several times, etc.

Other problem of this program is that we cannot change the behaviour of it in certain conditions. If you change the value of the numbers in the program in a way you make the system irresolvable the program will fail. Test it, change the values of a, b and c to 1 and d, e and f to 2. This system has infinite solutions and will make the program to fail. If you compile and execute with the new values, you should get something like the following:

```
$ ./main.exe
1x+1y=1
2x+2y=2
x = -nan; y = -nan
```

What in tarnation is a nan? It is one of the special values that decimal numbers in C can hold (remember IEEE 754). It means "Not and Number". So, the result of that operation is not a number. How is that possible? Because we have divided by zero. If you know a little about calculus you'd know that a number divided by zero is an indetermination, that is, we do not know what it is. That's how C deals with that. And we are lucky, if instead of a decimal division it were an integer division, the program would simply close abruptly. It would be interesting to check if the system has a solution, and, if it had, then calculate it. This is done with control structures, which will be explained in the next chapter.

5.   Changing the normal flow of the program

As I introduced in the last section, it is convenient to be able to make the program do one thing or the other according to a condition. Also, it is possible (as a matter of fact it's essential) to repeat instructions according to conditions. This is called altering the flow of the program, because instead of execute one line after he next, the computer can jump to a place to another, either forwards or backwards.

5.1.   Conditional sentences

Conditional sentences are the ones that allow us to make the program flow to **diverge**. You are going to understand it easily with the next diagram.
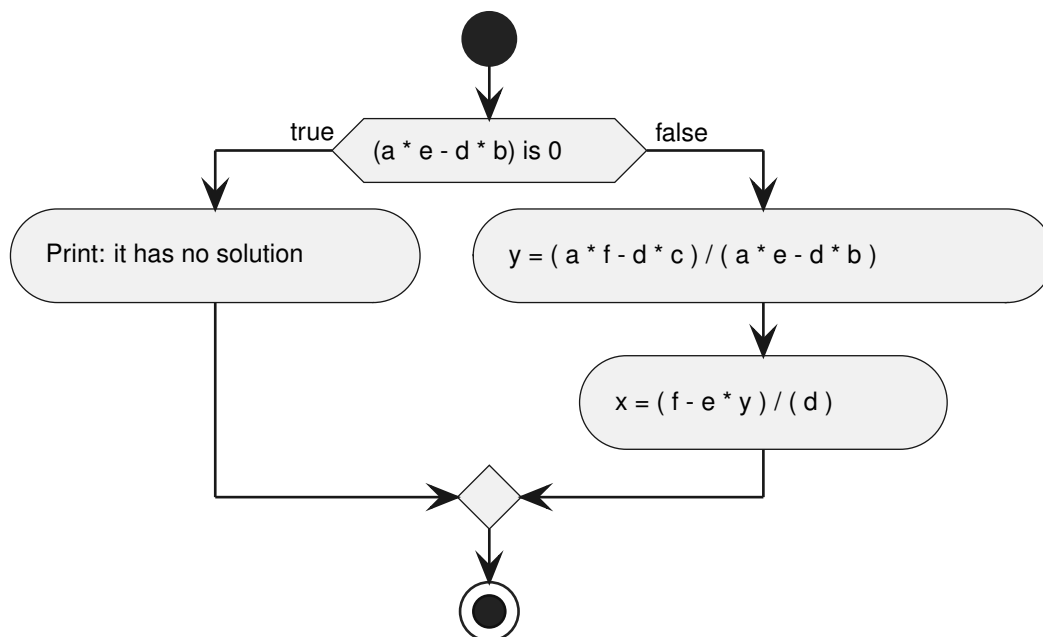
Figure 3: Flow diagram: solving an equation system with a conditional

What you're seeing is called a flow diagram, despise the fancy name, it is very easy to read. You start in the black dot in the top of the diagram, and you follow the arrow, a diamond means a **decision**, according to what is inside the decision (a condition), one branch or the other will be taken. If you read the text of the condition inside the diamond, I have written there what makes our system **irresolvable**. If you read the text next to the arrows that come out from the diamond, you will see that if $ae - db = 0$, we will print that the system is not solvable and exit normally. If it is not zero, we will continue doing what we were doing before.

5.1.1.   Logic operations

This drawing is nice, but let's see how we do that in C. Now I have to present to you another set of operators, called boolean operators or logic operators. In the section 1 I mentioned logic as a science. Concretely we are going to apply propositional logic, or first order logic. In this logic we have **facts** that can be only **true** or **false**, and they relate with each other with three operators. Let's see and example, and then we will hop to the theory.

Imagine a fire extinguishing system that works in this manner: "If the temperature is greater than 50 °C, the fire sprinklers will start working, if the temperature is lower, but there is smoke detected, the sprinklers will start working anyway". Since this is a science, let's write it in a formal manner. Each sentence that is conceptually different is a **proposition**, and they're generally named by letters of the alphabet from p onwards. Let's see which propositions we have:

1. Temperature is greater than 50 °C, let's call it p.

2. Smoke is detected, let's call it q.

3. The sprinklers go off, let's call it r.

$T > 50\,°\mathrm{C}(\mathbf{p})$	Smoke($\mathbf{q}$)	Sprinklers go off($\mathbf{r}$)
True	True	True
True	False	True
False	True	True
False	False	False

Table 6: Example of logic operations

In logic, there are thee basic operators:

1. Conjunction, commonly known as "and". It is written with the symbol $\wedge$.

2. Disjunction, common knows as "or". It is written with the symbol $\vee$.

3. Negation, commonly knows as "not". It is written with several symbols, for example $\sim$ and $\neg$, but it is written also putting a bar over the negated expression, for example $\overline{p}$.

$\mathbf{p}$	$\mathbf{q}$	$\mathbf{p \wedge q}$
False	False	False
False	True	False
True	False	False
True	True	True

$\mathbf{p}$	$\mathbf{q}$	$\mathbf{p \vee q}$
False	False	False
False	True	True
True	False	True
True	True	True

$\mathbf{p}$	$\mathbf{\sim p, \neg p, \overline{p}}$
True	False
False	True

Table 7: Tables of truth of the basic logic operations

The conceptual meaning of the operations are intuitive, but I will explain it here to tell some details. The conjunctios is true only when **both** of the operand are true, and is like joining two propositions with "and" in English. "It is raining and it's cold" is true only if at the same time it's raining and it's cold. On the other hand, disjunction is when you join two propositions with "or". "I have twisted my anckle or I have broken it", but there is a caveat here, if you look at the truth table you will se that when **both members** are true, the disjunction is true also. It is something that is not aligned with the spoken language, and you need to keep it in mind. Finally, negation is when you put "not" before the expression or preposition, "it is not raining" is true only if "it is raining" is false.

And, after this logical detour that would make Aristotle proud, what? We're almost there, in C the logic operators are written in this way:

1. Conjunction is written: `&&`.

2. Disjunction is written: `||`. (This symbols is called a pipe symbol, in the American keyboard you ir get pushing the key over the enter and shift.)

3. Negation is written: `!`.

If we go back to the example of the fire sprinklers, let's make a program that "simulates" the system, I will write it now. Up until now I have talked about true and false values, in C that is represented with an integer type (any of those available). Zero is the false value, and **all the other values** are true. Generally the result of a logic operation will be one if it's true, but you cannot assume that when using an integer type to store logic values.

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int temperature:greater_than_50 = 1;
5       int smoke = 0;
6       int aspersores = temperatura_mayor_50 || smoke;
7   }
```
Program 11: First program with logic operations

Yes, it looks like the boring programs we were doing up until now, do not worry, the matter is: if you go back to 6: Example of logic operations you're going to see that the program is just a disjunction, if there is smoke, or the temperature is high, or both, the sprinklers will go off. Nevertheless, this program is still a bit useless. We cannot check is the temperature is greater or lower than 50 °C, we have had to make it up. This leads me to present another set of operators to you, the comparison operators. This is easy, in C we can check is one variable (or expression) is equals, greater or less than another. I will write the table and let's improve the program we had before.

Operator	Description
<	True is the left operand is less than the right one, false otherwise.
>	True is the left operand is greater than the right one, false otherwise.
<=	True is the left operand is less than or equal to the right one, false otherwise.
>=	True is the left operand is greater than or equal to the right one, false otherwise.
==	True is both operands are equal, false otherwise.
!=	True is both operands are different, false otherwise.

Table 8: Comparison operators

Now you know comparison operators, let's make a program that checks is a given value is in the interval $(a, b]$, that is, between a and b, including b, without including a.

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int number = 5;
5       int a = 0;
6       int b = 10;
7
8       int is_in_interval = number > a && number <= b;
9
10      printf("%d is in (%d, %d] is equal to: %d\n",
11             number, a, b, is_in_interval);
12  }
```
Program 12: Primer programa con operaciones de comparación

As you can see, there is no need for parenthesis, comparations are executed before logic operations. I'll take advantage of the fact that I have already presented three set of operands (mathematical, logical and comparison) to tell you that there is nothing wrong with declaring variables along the program to hold partial values of what you want to calculate, specially if a given expression becomes too big. I would even advise you to do that when you feel like it and then remove the intermediate variables as an exercise, at least in your first programs.

5.1.2. Diverging the flow: the if

Now we know how to create logic conditions (propositions) that can be either true or false, we can create our first conditional sentence. In C, a conditional sentence is made with the key word if. I will present to you now the basic structure of an if, but this is **not** a valid C program.

```
1  if(/*condition*/)
2  {
3      //Executes only if condition is true.
4  }
5  else
6  {
7      //Executes only if condition is false.
8  }
```

Program 13: Basic structure of if sentence

Now we can improve our first program that simulates the sprinkler system! Let's add the comparison and logical operators.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int temperature = 25;
5      int smoke = 0;
6
7      if (temperature > 50 || smoke)
8      {
9          printf("Sprinklers activated.\n");
10     }
11     else
12     {
13         printf("Sprinklers deactivated.\n");
14     }
15 }
```

Program 14: Fire sprinkler program with logic and comparison operators

As we have been suffering before, each time you change any value you will have to recompile and execute. Change the values of the variable smoke and temperature to make the result of the conditional change. Finally, we can improve our program of resolution of linear equations, before performing any of the calculations we will check if the system has a solution. To do that, we need to check that the divider is not zero. To do that we simply declare more variables and then divide by them.

```c
#include <stdio.h>
int main(void)
{
    int a = 1;
    int b = 1;
    int c = 1;
    int d = 2;
    int e = 2;
    int f = 2;

    double divider = (a * e - d * b);

    if (divider == 0 || d == 0)
    {
        printf("The system is irresolvable.\n");
    }
    else
    {
        double y = (a * f - d * c) / divider;
        double x = (f - e * y) / (d);
        printf(" %dx+ %dy= %d\n", a, b, c);
        printf(" %dx+ %dy= %d\n", d, e, f);
        printf("x = %f; y = %f\n", x, y);
    }
}
```

Program 15: Linear equation resolution program with conditional

This is more alike to a "real" program, we could say that we are already hands down on the matter. But the conditional has more things to it. You can chain them. Sometimes we want to check a chain of conditions one after the other and execute the instructions related to the first true condition. Imagine a program that receives the temperature and, depending on what it is, throws a message about the weather.

1. If it's 40 ℃ or more, print "It's hot."

2. If it's 35 ℃ or more, print "It's warm."

3. If it's 25 ℃ or more, print "It's a nice day."

4. If it's 10 ℃ or more, print "It's a little bit chilly here."

5. If it's less than 10 ℃, print "It's very cold."

I have written the sentences in that way purposely so you realize that, even when forty degrees is more than ten, we don't want to print that it's hot, it's a nice day and it's a little bit chilly here, only the condition that arrives first. To do so C gives us the if-else sentence, in which we "chain" an if statement to the else of the other conditional. In that way, only if the condition before was false, the next one will be checked and, if it is true, the instructions inside that if-else will be executed, let's see it with an example.

```c
#include <stdio.h>
int main(void)
{
    int temp = 10;
    if (temp >= 40) {
        printf("It's hot.\n");
    }
    else if (temp >= 35) {
        printf("It's warm.\n");
    }
    else if (temp >= 25) {
        printf("It's a nice day.\n");
    }
    else if (temp >= 10) {
        printf("It's a little bit chilly here.\n");
    }
    else {
        printf("It's very cold.\n");
    }
}
```

Program 16: Example program for `if-else`

As you can see, we have finished the chain of `if-else` with an `else`, this means that in case no condition is true in the chain, the instructions inside the else will be executed. In this case, print "It's very cold".

Now we have seen conditionals, we can revisit our program that solves linear equation systems. If you remember, in the last revision (Linear equation resolution program with conditional) we deemed the problem as insolvable if $d = 0$. Nevertheless, if you remember a little of algebra from high school and you come back to the equation, you will see that d being equal to zero does not mean the problem has no solution, but that we should solve for y in both equations and not x. That is:

$$(1)\ ax + by = c \rightarrow y = \frac{c - ax}{b}$$

$$(2)\ dx + ey = f \rightarrow y = \frac{f - dx}{e}$$

$$(1)\ \text{and}\ (2) \rightarrow \frac{c - ax}{b} = \frac{f - dx}{e} \rightarrow ec - eax = bf - bdx \rightarrow x = \frac{bf - ec}{bd - ea} \rightarrow y = \frac{c - a \cdot \frac{bf - ec}{bd - ea}}{b}$$

In case this is not possible, it means that nor x or y are in both equations, that is, we have one equation that solves for x and other for y. Either we have $\begin{cases} ax = c \\ ey = f \end{cases}$ or we have $\begin{cases} by = c \\ dx = f \end{cases}$. Simply checking if $a = 0$ we will know in which case we are. Finally, we could be in the case that we had just one equation, if the terms that multiply x or y in any of them were zero, but in this case this is not a system and it is not possible to give values to x and y. If we write the code, let's see how it would look.

```c
1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 12, b = 2, c = 10, d = 50, e = 11, f = 17;
5      int irresolvable = 0;
6      double divider, x, y;
7      if (a != 0 && d != 0) {
8          divider = (a * e - d * b);
9          if (divider == 0)
10         {
11             printf("The system is irresolvable.\n");
12             irresolvable = 1;
13         }
14         else
15         {
16             y = (a * f - d * c) / divider;
17             x = (f - e * y) / (d);
18         }
19     }
20     else if (b != 0 && e != 0) {
21         divider = (b * d - e * a);
22         if (divider == 0) {
23             printf("The system is irresolvable.\n");
24             irresolvable = 1;
25         }
26         else {
27             x = (b * f - e * c) / divider;
28             y = (c - a * x) / b;
29         }
30     }
31     else if (a == 0 && b == 0 || d == 0 && e == 0) {
32         printf("This is not a system.\n");
33         irresolvable = 1;
34     }
35     else {
36         if (a != 0) {
37             x = (double)c / a;
38             y = (double)f / e;
39         }
40         else {
41             x = (double)f / d;
42             y = (double)c / b;
43         }
44     }
45     if (!irresolvable) {
46         printf(" %dx+ %dy= %d\n", a, b, c);
47         printf(" %dx+ %dy= %d\n", d, e, f);
48         printf("x = %f; y = %f\n", x, y);
49     }
50 }
```

Program 17: Program solving a linear equations system with conditionals

This program is a little long, but I am going to go conditional by conditional. First of all, let's circle back and rethink which possibilities we have according to the system. We can be in one of these four cases:

- x is present in both equations, therefore, we can solve for it in both equations as we did the first time we solved the problem. In this case, if the divider we calculated is zero, the problem is not solvable.

- y is present in both equations, we can solve for it as we did just before showing this version of the program.

- In one of the equations, both x and y are multiplied by zero, therefore this is not a system, and we cannot solve it.

- We can be in the case we can't solve both equations for one unknown, because one of them is present in the first equation, and the other in the second one.

In the first lines of the problem I am declaring a bunch of variables, I am sorry because I didn't teach you to do it in this abbreviated way, but I wanted the program to fit in one page. We will explain it later. I just declare the same values as before, and a logic variable called irresolvable, that will allow me to know if I have solved the problem at the end of it to print the solutions. The cases listed before are present in my program, in the first `if` we check if the coefficients that multiply x in both equations are different from zero. In this case, we try the first solving method: calculate the value of the divider and applying it. Inside this condition, if the divider is zero, the system is not solvable, else, we simply solve the system with it. As you can see, you can put conditionals inside conditionals.

In the next `if-else` we are checking if the can solve for y in both equations. This case is basically the same than the other, but applying the other set of formulas. We calculate the divider of the value of x in this case, if it is zero, we cannot solve it. The next conditional checks if in any of the equations the coefficients of both unknowns are zero. In this case, as we said, this is not a system and we cannot solve it. Finally, if we are not in any of the aforementioned cases, that means we're in the last one: x is in one equation and y in the other. We check if a is zero, if it is not, we know the first equation gives us the value of x and the second the value of y.

At the end, we simply check the variable irresolvable, that tells us if we have marked the program as so. If we haven't it means we have solutions, so we just print them.

5.2. Code blocks and scopes

Now you know the first control structure, I must talk to you about **code blocks**, a code block is the piece of code that is between two braces ({...}). The main implication of enclosing code between braces in a block is that the variables declared inside it are not visible outside, but those declared outside are visible inside the block. If you remember the basic structure of the conditional, you will see it includes braces, also, in the first program that I presented to you I told you you will always have to write a series of lines, those include a single code block. Inside that block we have put all the instructions in all our programs. Now we have conditionals, we have nested blocks (blocks inside blocks).

All variable declared in C has a **scope**, that is the portion of the code in which the variable can be seen, the scope of a variable is the block in which it's been declared and all those blocks inside that one.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int exterior = 0;
5      if (exterior == 0) {
6          double inner = 1.3;
7          exterior = 120; // OK: variable from an outter block
8      }
9      inner = 10.3; // Error: variable not defined
10 }
```

Program 18: Example of scope of declared variables

If you try to compile the code I just shown, you will se the compiler says the variable `inner` is not declared, even when you declared it "before". Another of the side effects of blocks is that you can define variables that already existed in outter blocks. Let's see an example.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int number = 0;
5      int exterior = 3;
6      if (number == 0) {
7          int exterior = 10;
8          printf("%d\n", exterior);
9      }
10     printf("%d\n", exterior);
11 }
```

Program 19: Example of redefinition of variable

If you compile an execute this program, you'll see it prints firstly 10 and later 3. This is because there are two variables with the same name. How is this possible? How does C to which variable are you referencing? Simply those variables that are local (those declared in the block you're in) have priority over those declared in more external blocks. This means you cannot access external variables if you have declared any with the same name in an inner block.

Now that you know that the scope is, let's explain some things of those I couldn't explain before. Firstly I want to tell you that outside the braces of our first program we can write things. Concretely, we can declare variables, which are called **global variables**. They are called like that because they have a "global" scope. That is: any other instruction in your program has access to them (unless they declared a variable with the same name). I am not going to take much time in them, at the moment, they're not useful to us, but I am going to show you an example of how a program with them would look.

```
1  #include <stdio.h>
2
3  int globalVariable = 20;
4
5  int main(void)
6  {
7      printf("%d\n", globalVariable);
8  }
```

Program 20: Example of a program with a global variable

The rule is that variables must be declared at the start of the block. This is not mandatory, but it is a good practice. I leave it to your election but I would encourage it. Also, you must declare them in the most inner block in which you can, for example in the program 15 we have declared the variables x and y in the only block we needed them, we could have declared them at the start of the program, or as global variables, but, since it was not necessary, we didn't do it.

5.3.   Other jump onstructions: `switch` and `goto`

You know the most important way to make the flow of the program diverge, but there are other two that still have some utility, I have already named them in the title: `switch` and `goto`. The first one behaves like a distributor of the flow of the program, given a variable it examines the value and compares it with a series of cases, according to the case, it jumps to that line. We could imagine the `switch` like a factory worker that classifies products that go out an assembly line, according to the state of the product it does a thing or another.

```c
#include <stdio.h>

int main(void)
{
    int day = 3;

    switch (day) {
        case 0:
        printf("Todays is: Monday\n");
        break;
        case 1:
        printf("Todays is: Tuesday\n");
        break;
        case 2:
        printf("Todays is: Wednesday\n");
        break;
        case 3:
        printf("Todays is: Thursday\n");
        break;
        case 4:
        printf("Todays is: Friday\n");
        break;
        case 5:
        printf("Todays is: Saturday\n");
        break;
        case 6:
        printf("Todays is: Sunday\n");
        break;
        default:
        printf("That number is not any day!\n");
        break;
    }
}
```

Program 21: Example of a program with a `switch`

If you look the example program, a `switch` starts in a similar way to a `if`, but inside the parenthesis there is not a condition, but always a variable. That variable must be an integer type. After that, in the body of the `switch` there is a set of lines that start with `case`, after that word you must write a literal value (it cannot be a variable) and then the lines you want to execute in case the the variable in the switch has the value of this case. That set of instructions can end or not with an special instruction called `break;`. This instruction makes the flow of the program to exit the switch. We need this because if we didn't put it, the instructions in the following cases would execute also. If you execute the program as it is written, it will print only the message of the Thursday, but if you remove the breaks and recompile, it will print all days from Thursday on.

You would see there is a line that is not a `case`, but has been created with the instruction `default`. This is because this keyword allows us to tell what would happen if the value does not match any other case. In this example we will print the number is not assigned to any day. You may have noticed, or not, that a `switch` with `break` in all the cases is basically a chain of `if-else`. This is true, a `switch` can be always replaced with a chain of `if-else` (if it has break in every line). In this case, the code associated with the default would be the code in the `else` at the end of the chain.

The `switch` is an example of jumping to a label. A label is something special because it stablishes a point in the to which you can jump with or without a condition. The other instruction to jump to a label is `goto`, let's see a simple example and, later, I will explain its most common use case. The code example for that case will be introduced in a later section, because it will be difficult to understand without advancing more in the contents of the language.

```c
#include <stdio.h>

int main(void)
{
    printf("Hello, this is the start of the program.\n");

    goto final;

    printf("I am a line that should not be printed.\n");

    final:
    printf("We have ended the program.\n");

}
```

Program 22: Example of a program with `goto`

If you compile the program and execute it, you would see that it prints the first and last line, but not the second. This is so because we have used the `goto` instruction to jump to a later point in the program. The use of `goto` is very dangerous, in the sense that it shows you haven't thought properly your program or you lack tools to make it in a better way. Therefore, until we haven't gone through other sections of this manual, I would discourage its ussage. I advise you to save gotos as a novelty more than a common-use tool.

5.4. Repeat instructions: loops

At the start of this section I expalined to you that making the computer to repeat instructions is necessary to make programs work, and here we're going to learn how: with **loops**. There two main kinds of loops: `while` and `for`. In the same fashion we did with the conditional, we will see firstly the flow diagram of the structure, later how it is written in C and finally an example on how to use the loops in some programs.

5.4.1. The `while` loop

This is the most basic kind of loop, and that's why we're going to explain it firstly, it sets a series of instructions that will be executed as long (while) a condition is true. Imagine you wanted to print all numbers from one to 100. You could write 100 lines that print every number or you could tell the computer to print a variable, add one to it, and then print it again while it is less than 100.
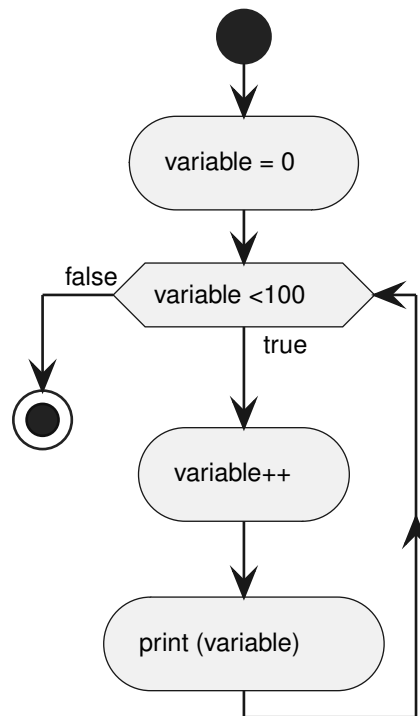
Figure 4: Flow diagram: program that prints the numbers from 1 to 100

As always, I leave you the code that would produce this result.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int variable = 0;
5
6      while (variable < 100) {
7          variable++;
8          printf("%d\n", variable);
9      }
10 }
```

Program 23: Example with a `while` loop

The `while` loop is the simplest of them, as you can see. The instructions that are in the loop are executed only in the condition is true. This has two implications: is the condition is **not** true when you arrive to the `while` instruction, the loop will never be executed, not even once. The other is that if there is nothing that changes the value of the condition of the loop inside it, the loop will execute forever. For example in this loop we change the value of variable so it reaches 100 and the loop ends.

There is a special kind of loop `while` that will execute the instructions always `at least once`, because the first execution will be done before evaluating the condition. Let's see an example.

```c
#include <stdio.h>
int main(void)
{
    int variable;

    do{
        variable = rand();
        printf("Variable aleatoria = %d\n", variable);
    }while(variable != 10);
}
```

Program 24: Ejemplo de programa con un bucle `do-while`

This program generates a random number (you must believe it until I explain more concepts) and prints it, then, if the number is not 10, prints it again and changes the variable value to another random value. Why would we need to use `do-while`? because with it we don't need to initialize the variable outside the loop, and all the ocurrences of the use a `rand` will be inside. This loop is much less used than `while` and shares with the switch that is it something that doesn't get used much, but when it does, it makes things much easier.

5.4.2. The `for` loop

The `for` loop is a loop that works like the while loop, but that does two things more: it executes an instruction **before executing anything inside the loop** and other **at the end of each repetition**. Let's see the flow diagram and how it's written.
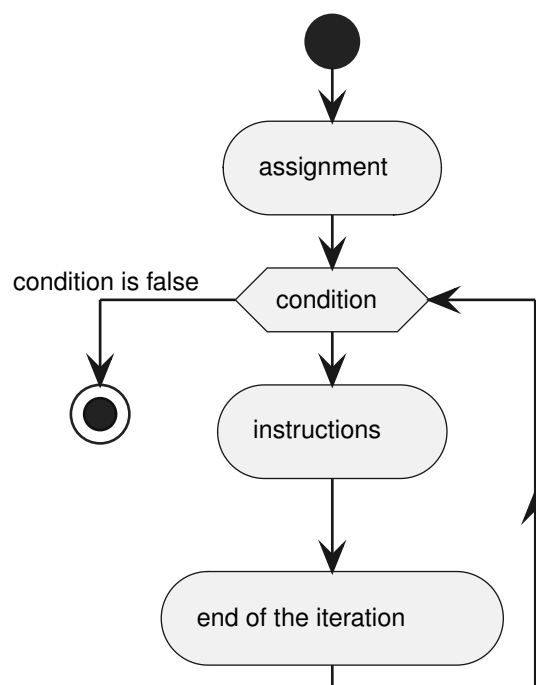
Figure 5: Flow diagram of a `for` loop

```c
1  #include <stdio.h>
2  int main(void)
3  {
4      for (/*Assignment*/;/*Condition*/;/*End of iteration*/) {
5          /*Instructions*/
6      }
7  }
```
Program 25: Structure of a `for` loop

Where I write assignment is because in that part of the loop you must write an assignment to a variable, or an assignment and declaration in the same place. Where I write condition you must write the condition that will control when the loop will stop executing (or if it executes at all). At the end of every repetition of the loop (formally we call iterations to the repetitions of the loop) the instruction you wrote where I wrote end of iteration will be executed. It's much clearer in the example.

```c
1  #include <stdio.h>
2  int main(void)
3  {
4      for (int ii = 1; ii <= 100; ++ii) {
5          printf("%d\n", ii);
6      }
7  }
```
Program 26: Example of a program with a `for` loop

This program performs the same task that we did with the `while` loop. As you can see, we declare and initialize a variable called `ii`, with value one, execute the body of the loop (the body is the block of code that will repeat itself) and, at the end to every iteration the variable will be incremented in one. This kind of loop tends to be very used along something called arrays, which we will see later.

5.5. Loop interruption

Sometimes we want to end an execution of a loop in the middle of an iteration, in this case we are going to write a program that calculates the power of two that is greater or equal than a target.

```c
1  #include <stdio.h>
2  int main(void) {
3      int steps     = 0;
4      int max_steps = 20;
5      int number    = 1;
6      int target    = 1024;
7
8      while (steps < max_steps && number < target) {
9          steps++;
10         number *= 2;
11     }
12     printf("2 to the %d is greater or equal than %d\n", steps, target);
13 }
```
Program 27: Example of interruption of a loop with an auxiliary variable

This program calculates the lowest power of 2 that is equal or greater than a target. We also make sure not to calculate any longer than the max number of steps that we want. The variables we need are: the counter of the steps we have done (`steps`) (the power we are testing), the maximun number of steps we are allowed to test (`max_steps`), then a variable called `number` that allows us to keep multiplying it by two and, finally, the `target` we want to check. So, basically we have a `while` with this condition: as long as the steps we have done are less than the max, and number is less than our target, we multiply number by two and increase the counter of steps.

```c
#include <stdio.h>
int main(void) {
    int steps     = 0;
    int max_steps = 20;
    int number    = 1;
    int target    = 1025;

    while (steps < max_steps) {
        steps++;
        number *= 2;
        if (number >= target) {
            break;
        }
    }
    printf("2 to the %d is greater or equal than %d\n", steps, target);
}
```

Program 28: Interrupción de un bucle con la instrucción `break`

In this case we have moved half of our condition to a conditional inside the loop, that checks if the number is greater or equal than the target. Mind that we have had to **invert** the condition. Before, we checked if number was strictly less than target, because it was the condition to **keep computing**, here we are checking the opposite, if number is greater or equal than the target, because this new condition **ends** the computation. If the breaking condition is true, we execute a `break`, exiting from the loop, printing the result and ending the program.

This way of writing loops is discouraged, ideally all the conditions that affect a loop termination should be in the loop, or in a variable that gets cheked in the loop itself. Hence this is not a good practice in programming, but this is not a manual about that, but to learn the language, I put this example to allow you to understand this if you see code where it appears.

Also: `break` has its counterpart, which is `continue`. This instruction omits **what is left of this iteration** and jumps directly to the next. Imagine, for example, we want to print if the years from 1 to 2021 are leap years or not. A year is a leap year given these conditions.

1. It is divisible by four.

2. It is not divisible by one hundred.

3. It is, in any case, if it is divisible by four hundred.

If one of those conditions is not true, there is no reason to check any other one, let's see how it will be written without `continue`.

```
1  #include <stdio.h>
2  int main(void)
3  {
4      for(int ii = 0; ii < 2021; ++ii){
5          int bisiesto = ii % 400 == 0 ||
6                          (ii % 4 == 0 && !(ii % 100 == 0));
7          if(bisiesto){
8              printf("The year %d is a leap year.\n", ii);
9          }
10     }
11 }
```

Program 29: Example of algorithm of leap year

A somewhat complex condition pops up, but if we write it with `continue`:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      for(int ii = 0; ii < 2021; ++ii){
6          if(ii % 4 != 0){
7              continue;
8          }
9          if(ii % 100 == 0 && ii % 400 != 0){
10             continue;
11         }
12         printf("El año %d es bisiesto\n", ii);
13     }
14
15
16 }
```

Program 30: Example of algorithm with continue

It uses more lines, but some people may argue it is easier to read, you could read it as "if it is not divisible by four, go to the next number, if it is divisible by one hundred but not by four hundred, go to the next too". Also, the printing order is not in a conditional, because if we reach the line, we already know the number is a leap year.

As it happened with `break`, this is not a better way to write a program, but I needed to invent some example with the limited content I have shown up until now. Again, take `continue` as a novelty more than something that will be used often.

6. Data structures

Data structures are one of the most important things in programming and in computer science, until now the only structure you knew were variables, each one of a type and with a different name. This is the simplest structure, but very often we need more complex structures, in this section we will see two simple data structures that C provides to the programmer: the array and the struct, or structure. Structure has a generic meaning as the one I used in the title, and the sepecific one of being the artifact of the C language I am going to show you. I will try to say structure when I mean any generic data structure and struct when I mean the C language feature specifically.

6.1. The array

Sometimes we want to pack data of the same type together, those packets are called arrays. An array is an structure in which we declare space for several variables, which we will reference by the name of the array and its position inside it. That is: we reference data in the array by "the fifth element in array a". Let's see how they're declared and used.

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int list_of_numbers[10];
5   }
```
Program 31: Example of declaration of an array

In the example we have declared and array with ten position of type `int`. And here there is the first important thing: the elements in an array do not start from one, but from zero. That means that an array like this one with ten positions hasn't got a position number ten, but positions from zero to nine. To access any element in an array we must write the name of it and, between square brackets (`[]`), the position we want to access. Inside the brackets there may be any expresion with an integer value, variables included, which is the most common.

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int list_of_numbers[10];
5       list_of_numbers[0] = 30;
6       int a = list_of_numbers[0];
7
8   }
```
Program 32: Example of ussage of an array

In this program we are declaring the array, assigning a value to the first possition and then using that value to assign that possition to another variable, the values of the first position of the array and the variable are not linked. If you remember what I said in the last section, we will use a lot of `for` loops alongside arrays.

```c
#include <stdio.h>
int main(void)
{
    int list_of_numbers[10];
    for (int ii = 0; ii < 10; ++ii) {
        list_of_numbers[ii] = ii;
    }
}
```

Program 33: Example of how to use an array

We must make several considerations, the variable you have declared in a `for` loop when you access an array tends to be called `i`. It is just a personal habit of mine to call it `ii`, you can do it as you wish, but if you want other programmers to read your code easily, I would encourage you to use any of those two alternatives; secondly: look carefully the loop, `ii` takes values from 0 to 9, which are the positions of the array, and it assigns to them the values of the variable we are using to access the array, so we wad number in it with values 0, 1... to 9.

But arrays can have more than one dimension, that is, if we now have arrays of data, we can have matrixes, or cubes, or even structures of unlimited dimensions. In general, human beings have problems managing more than two or three dimensions, so I will show you an example on how to use a bi-dimensional array.

```c
#include <stdio.h>
int main(void)
{
    int matrix[5][10];
    for (int i = 0; i < 5; ++i) {
        for (int j = 0; j < 10; ++j) {
            matrix[i][j] = 1 + (i * 5 + j);
        }
    }
    for (int i = 0; i < 5; ++i) {
        for (int j = 0; j < 10; ++j) {
            printf("%2d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

Program 34: Ejemplo de uso de array bidimensional

The first dimension of the array, in this case 5, inficates how many rows it will have, the second, the columns. You can think of array with two dimensions as an "array of arrays". This is expandable to all the dimensions, as we said before. You must be careful, if you try to access to an position of the array that does not exist it is very probable that your program ends abruptly. If you use nested loops with arrays of several dimensions, the custom is that the variables that go in the inner loops after `i` are called with the following letters (j, k...). This is a mathematical reminiscence, where big operators like summations or products of series use indexes with these letters.

6.2. The struct

In C a struct is a "pack" of data from different type that has a name. It would be like grouping a series of variables and refer to them as a set. Which advantage has this got? We can create new types sticking together the basic ones. If you think about that, it's something very natural, a struct allows us to use the basic construction blocks of the language, basic types, and create new concepts with them. In general, we want all the parts of our code to be able to see these new types, so we will declare them outside all code blocks, in tye same way I told you a global variable is declared.

When we create a variable of this new type, we are **instantiating** a struct. In computer science instantiate is creating a variable of a given type. We could say "instantiating an integer" but it is only applied to composed types, not basic ones. So, firstly we must create our new type and then variables of said type.

Now I present you the syntax to declare a struct, to create a variable of that type and use it.

```
1   #include <stdio.h>
2   struct my_struct {
3       int field1;
4       char field2;
5       double field3;
6   };
7
8   int main(void)
9   {
10      struct my_struct my_variable;
11
12      my_variable.field1 = 100;
13      my_variable.field2 = 'b';
14      my_variable.field3 = 3.3;
15  }
```

Program 35: Declaration, instantiation and use of a *struct*

The data that are inside a struct are called **fields**, and they are accessed with a dot. If you're thinking that makes the point an operator, you're right, dot is an operator that accesses to the fields in an struct. Notice that to declare a variables of the new struct type you need to put the keyword `struct` before, not only the name you gave to it.

Let's see an example of an use case of this. Imagine we want to make a program that calculates the distance between two points, we could do it this way:

```c
1  #include <stdio.h>
2  #include <math.h>
3  int main(void)
4  {
5      double point1_x = 1.1;
6      double point1_y = 3.2;
7      double point2_x = 2.3;
8      double point2_y = 5.4;
9
10     double diff_x = point1_x - point2_x;
11     double diff_y = point1_y - point2_y;
12     double distance = sqrt(diff_x*diff_x + diff_y*diff_y);
13
14     printf("P1 : [%f, %f]\n", point1_x, point1_y);
15     printf("P2 : [%f, %f]\n", point2_x, point2_y);
16     printf("Distance: %f\n", distance);
17 }
```

Program 36: Example of calculation of distance between points in a plane

As you can see, the program works and does what it is suppossed to do... but it is quite confusing, the only thing that "joins" together the coordinates of any point is the name of the variable, also, you need to declare many variables . I think you can see that the very moment we start managing many points in a program, this could be something very difficult to understand. Let's see what would come out of using our new knowledge of the structures.

```
1   #include <stdio.h>
2   #include <math.h>
3
4   struct point_s {
5       double x;
6       double y;
7   };
8
9   int main(void)
10  {
11      struct point_s A;
12      struct point_s B;
13
14      A.x = 1.1;
15      A.y = 3.2;
16      B.x = 2.3;
17      B.y = 5.4;
18
19      double diff_x = A.x - B.x;
20      double diff_y = A.y - B.y;
21      double distance = sqrt(diff_x * diff_x + diff_y * diff_y);
22
23      printf("P1 : [%f, %f]\n", A.x, A.y);
24      printf("P1 : [%f, %f]\n", B.x, B.y);
25      printf("Distance: %f\n", distance);
26  }
```

Program 37: Calculating the distance between two points using structures

As you can see, the program is much more clean. Also, I do not have to rely in the name of the variables to signify that something is a point, but I can rely in the tools the language offers me.

Note: These two programs use a library (we will speak about them later on) and need to be compiled with an special option, simply add −lm to the line you have been using to compile until now, it would end like: "gcc −o main.elf main.c −lm" (without the quotes).

6.3. Initialization lists

Both arrays an structs have a way to initialize them in a concise way. If you remember the chapter about variables, initializing a variables is give it value for the first time. This way of doing so to arrays and structs is the initialization list. Let's see an example.

```
1   #include <stdio.h>
2   struct point_s {
3       double x;
4       double y;
5   };
6   int main(void)
7   {
8       struct point_s punto1 = {1.1, 2.3};
9       int my_array[10] = {1,2,3,4,5,6,7,8,9,10};
10  }
```

Program 38: Initializing with brackets

In the case of the struct, the members are initialized in the order we declared when creating the new struct type. So in the example the x coordinate will be 1.1 and the second 2.3. In the case of the array we have filled all the positions, if we put only some, the remaining will be initialized to zero. Be careful, because the initializers with brackets are good only to initialize, if we added to the program a line such as: `punto1 = {2.2, 4.6}` the compiler wouldn't let you compile. The same would happen with the array. With the struct we can do something, initialize only the fields we want, omitting some, etc. This is done with a little confusing syntax. Let's use an example with an struct that stores a date. One of its variables is logic, tells if it is a leap year. It's something we can calculate, but it is not unusual to store data calculated of an struct on itself so you don't have to do it all the time.

```
1   #include <stdio.h>
2   struct date_s {
3       int isLapYear;
4       char day;
5       char month;
6       short year;
7   };
8   int main(void)
9   {
10      struct date_s moonLanding = {.month=7, .year=1969, .day = 20};
11      moonLanding.isLapYear = moonLanding.year % 4 == 0;
12  }
```
Program 39: Initializing a struct with brackets and field selection

As you can see, firstly we have initialized the fields we wanted and later we calculated what was remaining. Since we have said whose fields we wanted to assign values to in the initialization list, we didn't even have to worry about the order. As we saw before the rule to know if a year is a leap year is more complex but this is just an example.

I want to add as a conclusion that it is posible (and common) to declare arrays of struct and have struct that have arrays as fields. For example.

```
1   #include <stdio.h>
2   struct point_s {
3       double x;
4       double y;
5   };
6   struct triangle_s{
7       struct point_s points[3];
8   };
9   int main(void)
10  {
11      struct point_s points[2] = {{1.1, 2.3}, {4.5, 6.6}};
12      struct triangle_s triangly = {{{1.1, 2.2},{3.3, 4.4}, {5.5, 6.6}}};
13      triangly.points[0].x = 1.6;
14      triangly.points[0].y = 3.4;
15  }
```
Program 40: Combinación *struct* con array

Pay attention to the initialization list in line 12, in it the most external brackets indicate that we are initializing the struct `triangly`, after it, there is another level of brackets because we're initializing the member `points`, which is an array and, finally, for each element of that array we use an initialization list for each point. A clearer but longer way to write it would be:

```
1  struct triangle_s triangly = { .points = {{.x = 1.1, .y = 2.2},
2                                             {.x = 3.3, .y = 4.4},
3                                             {.x = 5.5, .y = 6.6}} };
```

I have to warn you about one thing, when you initialize an array with an initialization list you cannot define its size with a variable, that is: you must do it with a literal value. This has a very clear reason: if you initialize an array with a variable as its size, for example: `int array[var] = {1,2};`, the list will indicate that the array is at least, two positions long, but the compiler does not know how much var will be. If it's two or more, it is ok, but if it's less, what would the compiler do? Make an array of that length indicated by `var` and drop values from the list or ignore `var` and use the elements on the list? Mind you that the compiler cannot predict the value of a variable until the program is executed. You may think that in simple programs it may do it, and yes, it could, but even then, it would be very chaotic to have a rule about how to create arrays that applies only in certain conditions that are not clear. So, keep this in mind, if you want an array with a concrete size, determined by a variable, do not initialize it with an initialization list. In any case, the compiler would tell you that it's wrong.

6.4. Exercises of the section

Now you already have significant knowledge I will propose some exercises to you to check you understood the concepts presented so far.

Ex. 1: Write a program that declares a struct that defines a circle in two dimensions (center and radious). Make a program that declares a variable of that type and calculates its area.

Ex. 2: Write a program that, using the struct point presented in the example, declares an initializes an array of them and prints the directions you must follow from a point to the next, for example, if the points were: $(1, 2), (3, 4), (2, 5), (2, 1)$ this will be printed:

```
Right, Up
Left, Up
Still, Down
```

Ex. 3: Make a program that declares a bidimensional array and prints the sum of its rows and columns in this way:

```
1 2 3 = 6
1 4 2 = 7
5 3 4 = 12
-----
7 9 9
```

If you put numbers with a different length the columns will be misaligned, you must not worry about this.

Ex. 4: Write a program that does the following for the number from one to 100, both included: if the number is divisible between two, you must print "fizz", if it is divisible by five, "buzz", and if its divisible between the two: "fizzbuzz", it shall not print anything otherwise. I leave here an example with the first ten numbers.

```
fizz
fizz
```

```
buzz
fizz
fizz
fizzbuzz
```

7.   Functions

We are arriving to one of the most important parts of the manual. As you have have read in the title of the section, we are going to talk about functions. In programming, a function is almost a direct translation of what a function is in mathematics. Let's start there, in mathematics a function is an object that receives a series of arguments from certian domains and gives a result that is in some codomain. In other words, it receives a set of mathematical objects each one from a type (numbers, vectors, matrixes...) and gives you a result of some type of mathematical items.

$$f(x) : \mathbb{R} \longrightarrow \mathbb{R}_{\geq 0}$$

$$x \to x^2$$

That thing up there is basically a very formal way of writing $f(x) = x^2$ and x is a real number. We have defined the domain (the real numbers) and the codomain (the **possitive** real numbers) and the transformation you need to make to the arguments to get the result. A function in C is the same, it is a piece of code that receives a series of arguments and "returns" a result. I am going to explain it in yet another way: a function is like a "black box" that receives ingredients and gives us a result, without the need to know what is inside.

After that introduction, let's see how they are declared, defined and used. The declaration is written as it follows:

1. Data type of return of the function, that is, the codomain, el data type the function is going to give us. For example, `double`.

2. Name of the function, in the same fashion than variables, functions have to have a name. For example: `power`

3. Opening parenthesis (`(`)

4. List of parameters (arguments) of the function, with their type, separated by a comma each one. In this case as example, two integers: `base` and `exponent`.

5. Closing parenthesis (`(`)

6. As always, end the line with semicolon.

Following the examples we set before, it would be like:

```
1  double power(int base, int power);
```
Program 41: Function declaration in C

But this function cannot be used, because we haven't defined it. Define a function is in a way like initializing a variable. And it is saying **what the function actually does**. To do that, we copy the definition of the function (without the semicolon) and we write a block of code. Which would look like this:

```
1  double power(int base, int exponent)
2  {
3      //Here you would put instructions
4  }
```
Program 42: Definition of a function in C

If you have been paying atention, you may have seen a similarity between that last fragment of code and the first program we wrote. This is not by chance, when you wrote `int main(void)`... what you were doing was **declare and define** a function called `main`. What you may be wondering now is why. This is because in the Linux operating system and in most operating systems the way programs start is by the operating system calling their !main! function. That is why I had to make you write all those lines without telling you what they were. Finally, the "shape" of a function (return type and type of arguments) is called signature of a function.

Let's come back to the function `power`, now we have it ready to be written, we can define its behaviour. In a function, the block of code that goes after the argument list is called **body**. Inside the body the arguments can be used as local variables. With this you can calculate the value you want the function to **return** and make C do so with the `return` (makes sense, doesn't it?). When you use this word, the function ends executing (you go out of it). Let's see how we can implement the function of the example that, if you didn't guess it by the name, calculates the power `base` to `exponent`. It would end up like this:

```
 1  double power(int base, int exponent)
 2  {
 3      double res = 1;
 4      int ii = 0;
 5      while (ii != exponent) {
 6          if (exponent < 0) {
 7              res /= base;
 8              ii--;
 9          }
10          else {
11              res *= base;
12              ii++;
13          }
14      }
15      return res;
16  }
```

Program 43: Example of a function in C

The function does this: declares a variable called `res` which will be where we will save the power we calculate. After that, we will use a `while` loop to multiply of divide (according to if the exponent is possitive or negative) as many times as it's needed. At the end, we will return the result.

All right, now we have defined the function let's see how you use it. To use a function you will "invoke" it. To do so we will simply write its name and the arguments it needs. Let's see an example, if before in our program we have defined the function, this fragment of code will print several powers. If you think about this, using a function in a programming language is like writing the concrete value of a function in mathematics, if $f(x) = x^2$ then you know writing $f(3)$ is the name as writing a nine.

```
1   #include <stdio.h>
2   // Paste here definition of power
3   int main(void)
4   {
5       double powers[3];
6       powers[0] = power(2, 10);
7       powers[1] = power(2, 0);
8       powers[2] = power(10, -3);
9       for(int ii = 0; ii < 3; ++ii){
10          printf("%f\n", powers[ii]);
11      }
12  }
```

Program 44: Invocación de función en C

As you can see, the values returned by functions are used as those returned by operators, you can save them in arrays or assign them to variables. And, now we're here, let's expalin another thing, as you may have already guessed, `printf` is a function. What is particular about it is that it's an special function, its first argument (the format) is a pointer, which a new type I haven't explained to you before. At this level, all the functions we write will have a fixed number of arguments, which is the more common thing.

And, lastly, the keyword `void`. Void means and empty space. This is the word we use when we want to indicate that a function doesn't need any argument, for example, `main`, or that it does not return any value. One second, what is a function that doesn't return anything good for? Well, while functions in C are very much like mathematical functions, they're not exactly the same, because functions in C can manipulate other things that are not their arguments, the global variables. The most common example is `printf`, that does a thing which is not returning anything, but manipulate the terminal, which is symbolized as a global variable of a concrete type. I told you when I mentioned them that they weren't useful for us "yet". When you have several functions, sometimes you need to use global variables because they will be available in all the functions, but, as I told you, it is not the best thing to do.

Finally, one precission: I have said to you that inside the body of the function the arguments behave like local variables, and that may have led you to this question: if you change the value of any argument, is that variable changed outside the function? No, the arguments of a function are **copies** of those that were given to the function in the call. Let's see it with an example:

```
1   #include <stdio.h>
2
3   void doble(int a){
4       a = a * 2;
5   }
6   int main(void)
7   {
8       int number = 3;
9       printf("%d\n", number);
10      doble(number);
11      printf("%d\n", number);
12  }
```

Program 45: Demostración de que una función recibe copias de sus argumentos

If you compile and execute this you'd see that both times the program prints a three. This is because, as I said, the arguments are copies of the ones you passed them.

7.1. Separation between declaration and definition

I have already explained to you how to declare and define a function separatedly, but in all the examples I have always included only the definition. This is because in the definition we include the declaration. There are two reasons to do both things separately, the first one is that you want to separate your code in several files, thing we will do later on, but the second is that you need to define all the functions because they use each other, let's see an example of two functions, whose target is to print always "Duck season", and "Rabbit season", depending on which was called first, the order will be one or the other. The code would be something like this:

```
1   #include <stdio.h>
2
3   void ducks(void){
4       printf("Duck season!\n");
5       rabbits();
6   }
7
8   void rabbits(void){
9       printf("Rabbit season!\n");
10      ducks();
11  }
12
13  int main(void)
14  {
15      ducks();
16  }
```

Program 46: Declaration not separated from definition

But if you try to compile this, the compiler will say that the function `rabbits` is not declared when you use it inside the function `ducks`, and, as you can see, it is not, because it is defined below that point. The program, nevertheless, works, but it is not, again, sensible, using functions without defining them before. How could we solve this? Putting the declaration of both functions **before** of their definitions. That is:

```
1  #include <stdio.h>
2
3  void ducks(void);
4  void rabbits(void);
5
6  void ducks(void){
7      printf("Duck season!\n");
8      rabbits();
9  }
10
11 void rabbits(void){
12     printf("Rabbit season!\n");
13     ducks();
14 }
15
16 int main(void)
17 {
18     ducks();
19 }
```

Program 47: Declaration separated of definition

If you execute this, your program will be executing forever, so hit control key and C at the same time to end it.

7.2. The functions and the arrays

The arrays are a special question when they're mixed with functions for two reasons: when you pass an array as an argument to a function, **you can modify the contents of it**. This is because arrays that are passed to a function turn into **pointers**. As you can see, I have mentioned them several times. They're one of the most central elements of the language and where its power is, hence I will explain them later on, but its presence is felt in an invisible way in all we are learning up until now, as you will see when we reach that point. At the moment, just remember that when an array is an argument of a function, you can modify its elements. Let's see an example of two functions, one modifies the elements of an array passed as argument and the other does not.

```c
1  #include <stdio.h>
2
3  void print_array(int array[], int array_size) {
4      for(int ii = 0; ii < array_size; ++ii){
5          printf("%d ", array[ii]);
6      }
7      printf("\n");
8  }
9
10 void add_one_to_each(int array[], int array_size){
11     for(int ii = 0; ii < array_size; ++ii){
12         array[ii]++;
13     }
14 }
15
16 int main(void)
17 {
18     int my_array[] = {1,2,3,4,5,6,7,8};
19     print_array(my_array, 8);
20     add_one_to_each(my_array, 8);
21     print_array(my_array, 8);
22 }
```

Program 48: Array use with functions

As you can see, if you execute this, you would print first the original array and later the array with its elements incremented in one. This shows you that elements in an array can be modified by functions. May be you wonder if you can modify structures. **No**, you cannot modify struct fields when you pass them as arguments. I also want you to see that when you pass an array to a function in the list of arguments you must write: `type name[]`, for example in both functions shown before: `int array[]`. Also, a function **cannot return an array**, this is for reasons I will explain when we go deeply into topics of pointers. Finally, functions cannot receive bisimensional pointer, for similar reasons.

7.3. Exercises of the section

In this section I will ask you to write several functions, needed to say that, even when it is not asked for in every exercise, you should check you have done it right, **testing** the functions calling them in your function `main` with values and printing the results.

Ex. 5: Write a function that tells if a number is prime or composed. Note: a number is prime only if it is divisible only by itself and the unit (1). One itself is not composed nor prime.

Ex. 6: Write a function that calculates the distance between two point structures in the last section. To calculate the square root of a number you must use the function `sqrt`, to be able to use it you should include at the begining of your program (just right under the line that says: `#include zstdio.h>`) the line `#include <math.h>` and add `-lm` to the command to compile the programs, that would be: `gcc -o main.elf main.c -lm`

Ex. 7: Write a function that receives an array of integers and a separator char that will print the elements in the array separated by that char, for example, to the array `{1,2,3,4}` and the char `'\n'`, it will print:

```
1
2
3
4
```

Ex. 8: Write a function that encapsulates the program 17: Program solving a linear equations system with conditionals. The function must receive the coefficients if the equations (a, b, c, d, e y f). It can receive them separatedly of in an array. To return the result you can create a struct that has two `doubles`.

Ex. 9: Write a function that normalizes the elements of an array of `double`. Normalize is expressing all the elements in terms of the unit, so to normalize it you must divide all elements by the biggest element

8.   The Memory

This is one of the most important sections of the manual. Even when this is a manual about C programming, it is very difficult, if not impossible, to program in C in a sophisticated way without understanding, at least partially, the memory of a computer. Up until now I have said that the variables that you declare, both of basic types and arrays or structs are stored in "memory", but, what is exactly a computer's memory? Well, let's start from the most evident thing, how does it look?
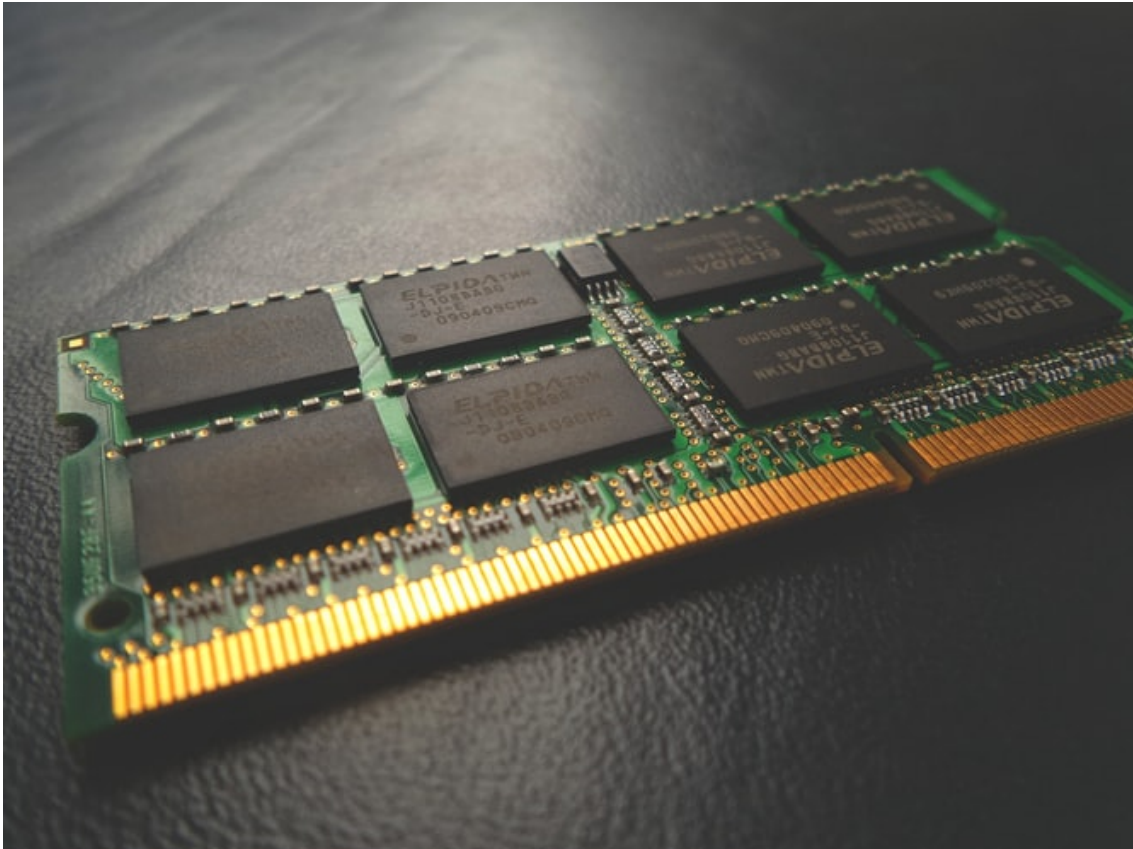
Figure 6: RAM memory module

Here you can see a module of memory of a computer. This module is a circuit board with memory chips soldered to it. The chips are the black rentangles. In them it is stored the information in binary code. A memory is a system from which you ask for a portion of information and it deposits the value in a set of "cables" that are in the computer, from which the values travel to the CPU, the processor, the memory is **addressed**. This means that each portion of the memory is referenced by a number.

Using an analogy, imagine that the memory is a notebook with a grid, in each cell you can store a figure or a letter, to be able to fill or read the cell, what we will do is assign a number to each one. We will start with the number zero and after that the one, two...

f	a	3	7	J	n	c	C	H	r	y	D	s	b	z
a	y	y	4	c	B	W	D	x	S	2	2	J	j	z
h	M	a	i	R	r	V	4	1	m	t	z	x	l	Y
v	K	W	r	O	7	2	t	K	0	L	K	0	e	1
z	L	O	Z	2	n	O	X	p	P	I	h	M	F	S
v	8	k	P	0	7	U	2	0	o	0	J	9	0	x
A	0	G	W	X	I	I	w	o	7	J	4	o	g	H
F	Z	Q	x	w	Q	2	R	Q	0	D	R	J	K	R
E	T	P	V	z	x	l	F	r	X	L	8	b	7	m
t	K	L	H	I	G	h	I	h	5	J	u	W	c	F

Table 9: Example of a grid with values

In the table you can see an emulation of this notebook, imagine that I tell you "tell me what is in the cell number 24". Keeping in mind that it is a table of 15 columns and that **we start counting from zero**, you'd have to say "S", because the **address** 24 is the nineth position of the second row. In general, computer memories are **addressed by bytes**, this means that for each byte there is a number (an address). You may have noted that it is absurd to draw this in a table if we are assigning simply numbers, this should be a continous list, a table of a single column. If you have thought that, you're right, because that's how memory is usually represented.

To sum it up: the memory in computers is a continous succession of bytes numerated from zero and on, to which we can reference by that number both to read and write.

8.1. Positional numerical systems: decimal, binary, hexadecimal

As we said in the introduction, the computer only understands binary code, and this is applicable both to the addresses of the memory and its contents. Due to this, I am compelled to teach you how the binary code works. The binary code is a positional numerical system. In a positional system, numbers are composed by figures, each one of those figures has a value depending on which **position** they occupy in the number. The more to the left they are in the number, the more they add to it. Remember when we learnt how numbers work, you had units, tens, hundreds, thousands... and so on and so forth. The system we use to write numebers is, therefore, positional, and decimal, because we have ten figures (from zero to nine). In binary it's the same, but with only two figures.

A positional numerical system works in this way: if the numeric base (number of different figures available, in the case of our system, 10) is n, each figure adds to the number the value of the figure multiplied by n to the power of how many figures remain to the right of this one. As always, let's see it with and example: if we write the number 34,789, the calculation we perform to know how much it adds up is this one (remember that any number that is not zero powered to the zero power is one):

$$3 \cdot 10^4 + 4 \cdot 10^3 + 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0$$

$$3 \cdot 10000 + 4 \cdot 1000 + 7 \cdot 100 + 8 \cdot 10 + 9 \cdot 1$$

Again, we can express it like tens, hundreds, units... which are the names we have to those concrete powers of ten.

So, if you had a binary number, you'd make the same calculation, but instead of unit, tens, hundreds and thousands, etc. you'd have to use units, couples, quartets, octets and groups of 16 (there is no word for that). Given the binary number 1110101, the calculation would be:

$$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 117$$

Now you already know how to read a binary number, you need to see how a decimal number is converted to binary. To do so, you do the following:

1. Divide the number by two, calculating the residue.

2. If the division leaves no residue, you write a zero on the left side of the resulting number, if it has residue (it can only be one), you write one.

3. Repeat until the **result** of the division is zero.

Let's see an example of this, given the number 253, if you apply the procedure, you should have something like the operations below. If you note the residues of the divisions in the direction of the arrow in the bottom of the diagram you could compose finally the binary number: 11111101.

$$
\begin{array}{cccccccc}
253 & \underline{|\,2} \\
5 & 126 & \underline{|\,2} \\
\underline{13} & 0 & 63 & \underline{|\,2} \\
1 & 6 & 0 & 31 & \underline{|\,2} \\
 & 0 & 3 & 1 & 15 & \underline{|\,2} \\
 & & 1 & 11 & 1 & 7 & \underline{|\,2} \\
 & & & 1 & & 1 & 3 & \underline{|\,2} \\
 & & & & & 1 & 1 & \underline{|\,2} \\
 & & & & & & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
$$

$\longleftarrow$
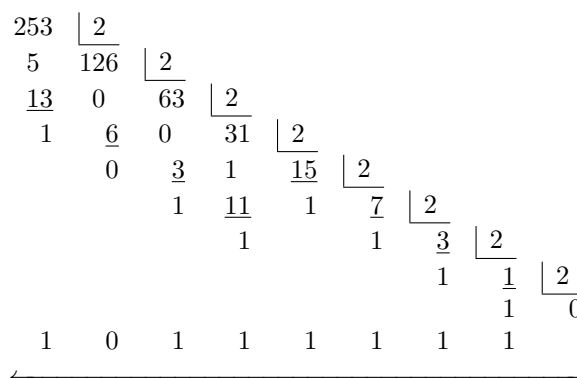
If you are an user of computers, you may have heard sentences like "this computer is a 32 bit computer" or "this computer is compatible with 64 bit software". That numbers of bits is the size of, between other things, the memory addresses. A computer of 32 bits has 32 bit memory addresses, hence it can address 2^{32} bytes. Newadays most computers are 64 bit, so most of them can address 2^{64} bytes of information theoretically. Of course, a computer will be always limited by the actual amount of memory it has which, in normal computers, is usually of a handful of gigabytes.

The problem is that 64 binary digits are too many to be read easily, look at this binary number: 1101001001010101001010100101101010101010101110100100111010010101. It's too long. Because of that, when memory addresses are written, they are written in a different numerical system. This numerical system is the **hexadecimal**. Is a system with base 16, with figures from zero to F. Yes, to F, you have read it right. Since numbers in our normal system have base 10, we haven't got symbols to represent a figure that has value 10, 11, 12... up to 15, so we use letter of the latin alphabet. Otherwise, it works in the same way than binary or decimal. A hexadecimal numbers is usually written with "0x" infront of it to tell the reader that what follows is a number in that base. Let's see an example, given the hexadecimal number 0xF2A.

$$0\mathrm{xF2A} = 15 \cdot 16^2 + 2 \cdot 16^1 + 10 \cdot 16^0 = 3882$$

To turn a decimal number into hexadecimal you must follow these steps:

1. Convert the number to binary

2. Divide the number in groups of four bytes **begining on the right side**.

3. Turn each one of the groups to decimal and write the corresponding hexadecimal digit.

Let's go back to the example of the number we converted to binary before, 253, in binary it is 11111101, if we wanted to convert it to hexadecimal we would need to divide it in groups of four: 1111 1101. the number 1111 is 15, so it'd be F, and the number 1101 is a 13, so it would be D, therefore 253 would be equal to 0xDF. If the leftmost group is not a 4 byte group, you must put zeroes on the left side. By the way, to convert from hexadecimal to binary, simply take every hexadecimal digit and convert it to binary again but, remember, each digit is a four byte binary number, so 0x33 would be 0011 0011. no 1111, that would be a different number entirely. You're prepared to start learning how the memory of a computer works.

8.2. The memory map

One of the most common way to represent the memory of a computer is with a **memory map**, that is a drawing in the shape of a column in which the contents of the memory are explained, indicating on the side the relevant addresses. Look at the following figure:
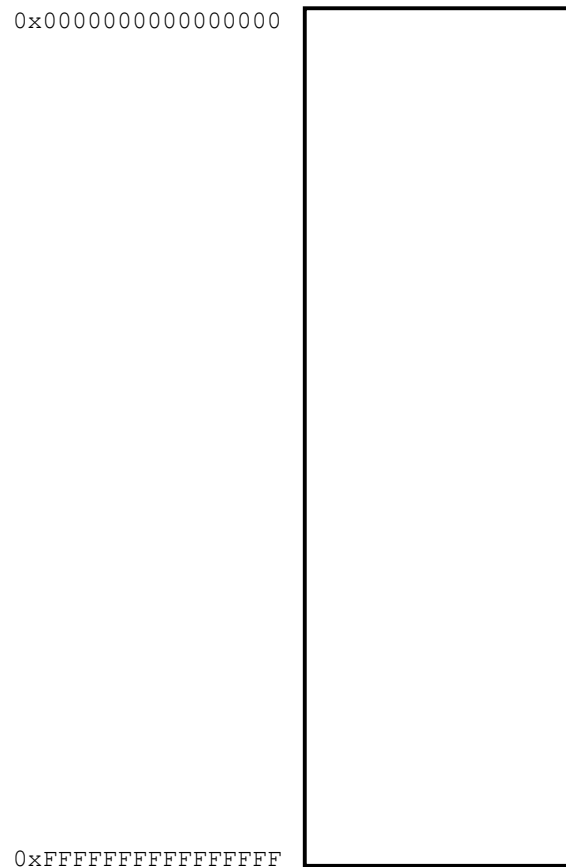
Figure 7: Mapa de memoria vacío

In that figure we have drawn the whole computer memory in a column, with the lower addresses (near zero) in the top and the higher ones (closer to 0xFF...) in the bottom. Generally I like more this representation, but in many sources and literature you'd see the map drawn in the other direction.

If your programs were the only software that executed in the computer, you'd have the whole map available for you and you wouldn't have to do anything to write on memory, simply... do it. But this is not the case, because your programs are executed thanks to the operating system. The operating system has many functionalities: it coordinates the programs that are executed in the machine, manages file systems, allows the CPU to understand devices such as keyboards and controllers... but one of its most basic functions is **memory management**. First of all: when a program is executed, its contents are loaded from where it is stored (your hard disk mainly) to your RAM memory, and a **process** is created. A process is the actual program running in memory, you could see the program as the blueprints of a car and the process as an instance of a car, concrete, that is working in the world. The OS gives processes memory blocks on which these can write or not, and **ensures** that they do not go out of their assigned memory.

In the section in which I talked about arrays I told you that if you accessed to a position of an array that didn't exist, your program would end abruptly, test it. Make a program that declares an array of, for example, 10 positions and afterwards writes something in position 2,500. You'd see how the program writes a message like this one when executed:

```
$ ./main.exe
Segmentation fault (core dumped)
```

It is possible that it does not throw the message, due to how the computer manages the memory internally. Anyway, for a C programmer, memory management (and specially checking that he does not write to or reads from memory blocks he shouldn't) is one of the most important tasks, if not the most. In this task, the operating system decieves our programs. To our program, you have an available memory that is the whole map (the 2^{64} bytes), even when the computer may have, for example, 8 GB which is several thousand times less. By the way, in general terms, each level in the prefixes of multiplication of 1,000 is not 1,000, but 2^{10}, 1,024, when you're talking about bytes, if we were exhaustive, we should write GiB, MiB and so on, which is the correct way to indicate those prefixes that multiply by 1,024. What the operating system does is to alocate what we ask from him in the **physical** memory and gives us addresses of that map of memory we think we have, and translates it. This process is the **memory virtualization**, and is one of the most important features of an operating system. Thanks to it, the programmer doesn't need to manage where the memory is physically allocated. Also, it allows each process to be isolated, the programmer has no idea which other processes are doing in their memory map, and other processes have no information about what we're doing with ours.

In this map of virtualized memory, which is efectively the one we're going to use, the operating system creates a set of **memory regions**, which are used to alocate different types of information of each program executing in a computer. Now I will show you a memory map with the most important regions, and I will explain what they are and why we care about them.

0x0000000000000000

Code (text)
Initialized data
Non initialized data
Heap
Stack

0xFFFFFFFFFFFFFFFF
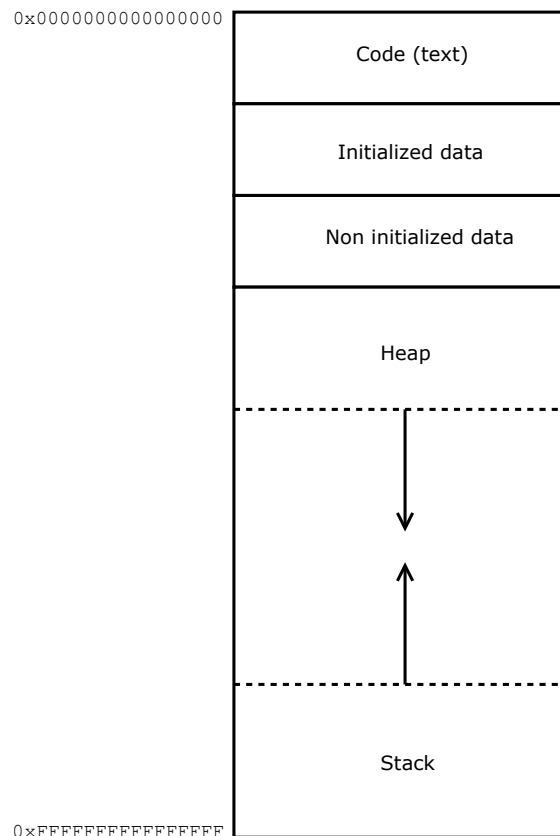
Figure 8: Regions of the memory map of a process

The text region is easy, it is were the instructions that your program is going to execute are stored. When I explained you what compilation was I told you we were going to transform our instructions in a binary the computer can execute. This is the section where it is stored. To generate a program, those instructions in binary are stored in the disk in the executable. When they load in memory, they go in this section. This section can't be directly read or changed by the program itself.

The next one starts to be interesting. This sections stores the value of the global variables, either of basic types, arrays, or structs that you write initialized in the code (initializing the variables from a function or other block of code won't count). This is because the values you store in them will exist before your program starts executing, in the very moment the process is created. The next section stores those **global** variables you haven't initialized. Why only the global ones in both cases? Because the function `main` is a function and the variables declared in functions (or any other block of code) are not stored here, but in the next section, the stack.

Look at the map, this section is at the bottom (towards the higher addresses), but I am going to explain it now because it is one of the most important. The variables you declare in a block of code: functions, loops, conditionals, etc. are stored. Why? Because in this section of memory data can go in and out, or better said, the memory of old data can be reused to write new ones. To explain how this is done, firstly I need to explain you how a stack (in general terms) works.

A stack is a data structure, like arrays or structs, which works in this way: when you put something in the stack, that element in on top, and when you pull out something from the stack, you can only pull out the last element, the one in the top of the stack. I am going to set a physical example: a can of the famous Pringles® potato chips. The only chip that you can pull out is the top one, if you want get more, you can, but always in the reverse order from how they were put in the can. Any stack works the same. The way this stack is applied to C programming is this way: when the execution flow goes into any code block, the variables declared inside it are **added** to the stack. This includes arrays and structs. When the flow goes out of the block, those variables are **pulled out** of the stack, that is, they're lost because it is understood they were already used. This is the very reason **functions cannot return arrays**. Because those array would cease existing once the function has returned.

Maybe you're wondering why you can declare basic types variables or structs inside functions and return them, if they're going to be destroyed once the function is finished executing. That is because, in the same way the arguments, the return value when it is not an array get copied. Concretely it left in the top part of the pile so the **code where it called** of the function can copy it with an assignation. With array **we cannot use the assignment operator**, it is not how arrays are copied. As a matter of fact the compiler would throw an error if you tried assigning an array to another.

Let's see an example of how the stack would look in the case of executing some of the example programs we've written. We're going to use the program 48: Array use with functions. In this example you're going to see that in the pile there two names (`my_array` and `array`), but remember they **point to the same array**. They only copy the address it starts, not the elements.

The stack starts empty

Going into `main`
my_array

Going into `print_array`
array
array_size
my_array

Going into `for`
ii
array
array_size
my_array

We exit the `for`
array
array_size
my_array

We exit `print_array`
my_array

Going into `add_one_to_each`
array
array_size
my_array

Going into `for`
ii
array
array_size
my_array

We exit the `for`
array
array_size
my_array

We exit `add_one_to_each`
my_array

Going into `print_array`
array
array_size
my_array

Going into the `for`
ii
array
array_size
my_array

We exit the `for`
array
array_size
my_array

We exit `print_array`
my_array

We exit `main`

Table 10: Example of the state of the stack in an execution

Now you know how the stack works and the implications it has in the arrays saved in it, we can see the last region and maybe the most important, the heap. This region stores the memory you ask from the operating system with a series of functions we're going to see. And you may be thinking: why would you do that if you can declare an array? Simple, this memory you ask the operating system for is always available to you **until you free it**. That means that, contrary to the arrays, it's your duty to worry about freeing it. Is one of the most important tasks of a C programmer, but to do so, you need to learn first what pointers are.

8.3. Pointers

Now you know that the memory is addressed by bytes, you must know how we use addresses in the C language. We do it with a new type (for us) called pointer. A pointer is a variable that stores a memory address, and allow us to communicate to functions or other parts of the program **a memory block**. You have already used pointers, but you didn't know what they were because I have chosen to explain other things first, although I have been anticipating its use.

I told you before that when a functions received an array as an argument it decays to a mere pointer. This means we do not copy the arrays functions receive, what we do is tell the function where the elements of the array are in memory. In this way, if it is necessary to perform a copy, we can do it, if not, the function can chose to manipulate or read them directly.

If a pointer symbolizes a memory address, you can think only one type for memory addresses is needed, but this is not the case. Each datatype (either basic or struct) has its own pointer, that is, there are not only pointers, but pointers to `double`, `char`, to this or that struct... Why is that? Because when using pointers with associated types, we know **what is** in the memory the address points to. For example, if we have a pointer to `int` that is 0xFB455DE, we know we must take that byte and the three next ones and decode them as an integer. Let's see it in the code.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 2;
6      int *ptr_to_a;
7      ptr_to_a = &a;
8      printf("a is in address %p and it is %d\n", ptr_to_a, a);
9  }
```
Program 49: Pointer declaration

In line 7 we declare a pointer variable for the first time, this is done with an asterisk that you see between the type of the variable and the name. This is where it is stablished that this pointer is associated to an `int`. In the next line we're assigning to this pointer the value of the address, `a` is allocated. We use the operator `&`. The name of the operator is ampersand. Said operator, infront of an expression, gives us the pointer to its type with the address said expression is stored in. Mind that, for this to work, this expression must be stored somewhere. That is, the temporal values would throw an error, for example: `&(a * 2)` would throw an error, because `a*2` hasn't been stored in any place. If you execute the program that uses the pointer, the output will be something like this:

```
$ ./main.exe
a is in address 0x7fffde738b6c and it is 2
```

Then, let's see a practical case of what pointers are useful for, for example: we have said several times that a function receives a copy of the arguments but, what if we wanted to save that effort? If, for example, you wanted a function that multiplies a number by other, maybe you do not want to copy it, the function returning the result and copying that result again in the variable, simply **leave the function work for you**. If we pass the pointer to our variable the function will be the one changing the value, let's see it.

```
1   #include <stdio.h>
2   void multiply(int* ptr, int b) {
3       (*ptr) *= b;
4   }
5   int main(void)
6   {
7       int a = 2;
8       int* pointer_to_a;
9       pointer_to_a = &a;
10      printf("a is in address %p and it is %d\n", pointer_to_a, a);
11      multiply(pointer_to_a, 4);
12      printf("a is in address %p and it is %d\n", pointer_to_a, a);
13  }
```

Program 50: Pointer ussage example

Here you can see you we declare a function that received a pointer to integer (the variable we want to multiply) and an integer (the number we want it to be multiplied by). In this function you'd see a new use of the asterisk operator, (*), which is the one for **dereferencing** a pointer. What is that? It is accessing the value that pointer is referencing (hence the name). Remember that, as a pointer, `ptr` stores the memory address, so we need a way to tell C to store in that address the multiplied number. To do so, we use the asterisk before the pointer. In simple words, the asterisk turns the `int*` into the `int` that pointer points to. It "follows the pointer". After that, we use the operator `*=` to multiply and assign. In line 11 you see how we simply call the function, without having to store what it returns (in fact, we have defined it as `void` so it does not return anything) and we avoid copying the integer we wanted to multiply.

Apart from the asterisk operator, there is another operator that is used with pointers that you must know, this is the arrow opeator `->`. It is used to access the fields of a pointer to a struct. This may be a little confusing, but I am going to take my time. Imagine we have the point struct we wrote in the section about structs. If, for any reason, we were using a pointer to it and wanted to access its fields, we should use the operator asterisk to dereference the pointer and then the operator dot to access the field. For example, let `point_ptr` be a pointer to `struct point_s`, to access its value `x`, we had to write `(*point_ptr).x`. It is not a problem, but I warn you that it is very common to have structs with pointers to other structs and so on... it can become pretty illegible in three or four times. That is why we have the arrow operator, to make the code before turn into simply: `point_ptr->x`. I want to clarify that this operator access to the field, does not gives us a pointer to said field. That is, following the example, `point_ptr->x` would be a `double`, not a `double*`. Later on we will see this operator in real use.

8.3.1. Pointer arithmetic

Arrays are in certain aspect (but **not** all) equivalent to pointers, due to this, pointers can be dereferenced with the square bracket operator. As a matter of fact, you can turn an array into a pointer explicitly in your program. As always, let's see how it is done:

```c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int my_array[10] = {0,1,2,3,4,5,6,7,8,9};
6      int* pointer_like_array = my_array;
7      for (int ii = 0; ii < 10; ++ii) {
8          printf("array[%d] = %d\n",ii, my_array[ii]);
9      }
10     puts("========");
11     for (int ii = 0; ii < 10; ++ii) {
12         printf("array[%d] = %d\n",ii, pointer_like_array[ii]);
13     }
14 }
```

Program 51: Arrays como punteros

What you see in the program 51 is what happens without you noticing it when a function receives an array, it's turned into a pointer and you can use it with the same operators of an array. This, nevertheless; is only valid for one dimension array, for a reason we will see later on. Exhaustively speaking, the square brackets operator is a *shortcut*. Actually what it does is add to the pointer and use the asterisk to dereference. When you add an integer type to a pointer, the pointer arithmetic starts to play, let's see an example and I'll show you how it works.

```c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int my_array[10] = { 0,1,2,3,4,5,6,7,8,9 };
6      int* pointer_like_array = my_array;
7      for (int ii = 0; ii < 10; ++ii) {
8          printf("In address %p there is a %d\n",
9              pointer_like_array + ii,
10             *(pointer_like_array + ii));
11     }
12
13 }
```

Program 52: Aritmética de punteros

If you execute the program you would see that the addresses are four units apart. This is because when you add an integer to a pointer, even when said pointer is a memory address of a memory addressed by bytes, due to being a pointer to **integer**, that expression of adding an integer to it is translated in adding the integer multiplied by the size of an int (four bytes). After this, we use the asterisk operator so this pointer we have added a pointer is dereferenced. Using pointer arithmetic is useful when you want to pass to a function the pointer of a position of an array. For example:

```
1  #include <stdio.h>
2  void multiply(int* number, int other) {
3      *number *= other;
4  }
5
6  void multiply_array(int* array, int array_length, int other) {
7      for (int ii = 0; ii < array_length; ++ii) {
8          multiply(array + ii, other);
9      }
10 }
11
12 void print_array(int array[], int array_size) {
13     for (int ii = 0; ii < array_size; ++ii) {
14         printf(" %d ", array[ii]);
15     }
16     printf("\n");
17 }
18
19 int main(void)
20 {
21     int array[] = { 1,2,3,4,5,6,7,8,9,10 };
22     print_array(array, 10);
23     multiply_array(array, 10, 10);
24     print_array(array, 10);
25 }
```

Program 53: Ejemplo práctico de aritmética de punteros

As an instance, if we use the function that multiplies a number without having to return it, we can write a function that does the same with an array (here you can appreciate how functions are a way to reuse code and avoid duplicating it). We can also see how, using pointer arithmetic, you do not need to have into account the size of each data type, the language does it for you. There is an alternative way to do this that you may see because it is more compact, and it is using the square brackets operator to get the element and then use the ampersand to retrieve the address, doing so, line 8 would turn into `multiply(&array[ii], other);`. In those cases, using one or the other is choice of the programmer.

8.3.2. The `char` **pointer**

Finally I am going to unveil uno of the misteries that I have been hiding from you for the most time (against my will, for the record) about the programs we have written up until now. This mistery is: what are those texts between double quotes, for example: `"Hello, world"`!. I got a bit ahead of myself in the title because I wrote the answer there, but they're an abbreviated way to write **arrays of** `char`. You know that a `char` is a letter, and that an array is a succession of data. The logic conclusion is that, in C, texts are `char` arrays. If they're `char` arrays, where is their declaration and why are they there between quotes. To sum it up: because we write texts in our program so often, the creators of C decided to add a **constant expression** to be able to declare arrays of `char` where it is needed, this expression is putting the text in quotes. There are expressions to declare arrys of other types, but they're not so important so we will see them in later sections.

Nevertheless; there is a differente between a `char` array (or pointer when it is passed to a function) and an array of another data type. The function `printf`, but in any place we say the size of the array, the number of elements on the memory the pointer points to. The functions that receive it must have some mechanism to know how many there are. That way is that every text chain (called strings in programming) in C finishes in a `char` with value zero. That is, when it is written `"Hola, mundo!"` we are creating in a quick way an array of `char` that contains **fourteen** `char`, the letters you can see and a `char` with value zero at the end. I am going to demonstrate this is the case with a little program:

```c
#include <stdio.h>

int main(void)
{
    char correct_string[] = "I like choccy milk.\n";
    char incorrect_string[] = {'I', ' ', 'l', 'i', 'k', 'e', ' ',
                               'c', 'h', 'o', 'c', 'c', 'y', ' ',
                               'm', 'i', 'l', 'k', '.', '\n'};
    printf(correct_string);
    printf(incorrect_string);
}
```

Program 54: Charr array

The first string will be printed correctly, but the second will print and, with all probability, after it, other characters will be printed, probably nonsense (execute the program a couple times, it may work well the first time). This is because since the second string is not ended in a char with value `zero`, `printf` doesn't know where to stop printing. Aside from this way of working with them, `char` arrays work like any other array and, when they're turned into pointers, like any other pointer. In later sections I will show how to manipulate text strings in more advanced ways.

8.3.3. The null pointer

There is an special value for all the types of pointer, which is the null pointer, or, as it is written in the language: `NULL`. It is a `void` pointer that is equal to zero. If you remember, the memory map that address would correspond to the text section, our program can't modify itself or read the binary instructions, among other things because part of that section is not our code, but the code of operating system which inserts itself in processes to allow us to do certain things. Hence the designers of the language used this value to symbolize a pointer that is in a special state.

One of the most important uses of this pointer is that it is used to express if an operation has gone well or not. For example, when we open a file with a function called `fopen`, this returns a pointer to a struct, if the file does not exist, or the program hasn't got permissions to open it, the pointer will be `NULL`. Many functions that receive a pointer use `NULL` to express an special behaviour. For ecample, let's write a function that encapsulates the functionality of the program 36, this has been an exercise, so if you didn't do it yet, do it now; but we're going to give it a twist: instead of receiving the structs, let's receive pointer to the structs. First, because as I said you, we save ourselves copying the structs and we can use `NULL` to indicate special values.

```
1  struct point_s {
2      double x; double y;
3  };
4
5  double distance(struct point_s *a, struct point_s *b) {
6      double res = 0.0;
7      struct point_s origin = { .x = 0.0, .y = 0.0 };
8      if(NULL == a){
9          a = &origin;
10     }
11     if(NULL == b){
12         b = &origin;
13     }
14     double diff_x = a->x - b->x;
15     double diff_y = a->y - b->y;
16     res = sqrt(diff_x * diff_x + diff_y * diff_y);
17     return res;
18 }
19
20 int main(void)
21 {
22     struct point_s a = {.x = 3, .y = 4};
23     double d = distance(&a, NULL);
24     printf("Distance: %f\n", d);
25 }
```

Program 55: Use of pointers to `NULL`

If you look at the function we have written, inside it we declare a point that symbolizes the **origin of coordinates**. What we do is that if one of the pointers to the structs is `NULL`, we understand that such point is the origin of coordinates. What we do is assigning to our arguments (which are pointers) the direction to this local variable we have declared. Remember: the arguments that a function receives are **copies** of the values. In this case, our argument is a pointer to a struct. Assigning to our argument other value **we are not altering the original structure**, because we haven't changed the value our argument points to, but the argument itself. Once this is done, we can calculate the distance in the same way we would do if they weren't pointers. What is the use of writing a function in this way? Finding the distance to the origin of coordinates is a common operation, doing things this way, we allow the program to call the function to do such calculation without declaring an extra struct to symbolize the origin.

8.4. Allocate and free memory

Now we have learnt what a pointer is, and to manage them, we can ask the operating system for memory of the kind that is stored in the heap and we can manage in a more flexible way than arrays. To do so we use two functions: `malloc` and `free`. The names of the functions are very descriptive, the first means "memory alloc" and the second frees the memory. When you need to reserve memory in the heap, you call `malloc` and ask for a contiguous memory block of n bytes. The function `malloc` returns a `void` pointer. That same pointer should, at some time, be freed passing it to `free`.

Talking about pointers to `void`, I told you that `void` means that the functions either don't receive anything or don't return anything. The meaning of a `void` pointer is related to that: it is a pointer that you do not know what it is, `malloc` has no way to know what you are going to do with the memory, therefore it returns a `void` pointer. A pointer to `void` is also useful when we want to write functions or structs that are compatible with different data types. Let's see first a simple example on how to use `malloc` and `free`.

One of the advantages of dynamic memory allocation occurs when other part of the program performs a calculation whose result has an unknown size. For example, imagine a function that, given an array of number, returns a vector with an instance of every distinct number, that is, erases repetitions, let's call it `erase_reps`. Functions cannot return arrays, that's something we already know, but the person that calls `erase_reps` could declare an array and pass it to the function, a problem arises nevertheless: we do not know the size that array will have. Is it true that we have an **upper bound**, if we give to this function an array of n positions, the result can only have, at most, n positions. So we could declare an array of n positions and tell the function to leave there the results. But there is a problem, how do we know which part is solution and wich part is excess? The only thing you could so is to return from `erase_packets` the size of the solution. In this case you'd have to an array of n positions of which you'd use a lesser amount. You'd be wasting memory and, even when it does not look like a problem, it may become one very quickly.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int *erase_reps(int* array, int array_length, int* final_length) {
5      *final_length = 0;
6      int preliminary_array[array_length];
7      for (int ii = 0; ii < array_length; ++ii) {
8          int unique = 1;
9          for (int jj = 0; jj < ii; ++jj) {
10             if (array[ii] == array[jj]) {
11                 unique = 0;
12                 break;
13             }
14         }
15         if (unique) {
16             preliminary_array[*final_length] = array[ii];
17             ++(*final_length);
18         }
19     }
20
21     int* result = malloc(*final_length * sizeof(int));
22
23     for (int ii = 0; ii < *final_length; ++ii) {
24         result[ii] = preliminary_array[ii];
25     }
26     return result;
27 }
28
29
30 int main(void)
31 {
32     int array[] = { 20,1,2,3,4,5,8,7,8,9,6,6,5,4,1,2,3,8,5,4,4,5,6};
33     int length;
34     int* result = erase_reps(array, 23, &length);
35     for (int ii = 0; ii < length; ++ii) {
36         printf("%d\n", result[ii]);
37     }
38     free(result);
39 }
```

Program 56: Example of dynamic allocation

In the line 2 of code we see how the line #include <stdlib.h> makes its appearance, it is needed to use malloc and free. After it we have the declaration of the function, we have made it return the pointer to the result and receive three arguments: the array from which we're erasing repetitions, the length of such array and a pointer to integer that will allow us to **indicate the length of the solution**. This pattern is very used in C programs, when you need the function to calculate a lot of things, you receive pointers to those things and write the results there. In the body of the function we assign 0 to final_length to start. After it, we declare a preliminar array to save the unique numbers, why an array? Because this is not the result, but an array we will use to save the numbers until we know how many there are, so we will assign to this array the upper bound I mentioned before, the list of unique numbers in an array can't be longer than the array itself. This is also a common pattern: using an auxiliary data structure that we will copy to another with a more proper size and definitive.

The next loop simply checks, for any element of the array, is that number appears before. Pay attention to the inner loop, for each i element of the array, we look at the elements before it (elements from 0 to i not including i). If it is equal to the one we're examining now, we use the variable `unique` to indicate if the number has been found before and, therefore, is not unique, so we assign this variable value 0. After that, once we have checked all the elements before the current one, we increment the final length and copy this number to the preliminar array. When the length is already calculated and all the numbers are in the preliminary array, we can use `malloc` to create the final solution.

The `malloc` return, as we have said, a pointer that indicates the start of the memory zone it has reserved for us. To do so, it receives the **size in bytes** we want. And you'll see here the sizeof operator. Yes, I have said operator, not a function. As a matter of fact, you may have noticed that no function is in blue in the code examples, and `sizeof` is. This is because it is a **unary** operator that gives us the size in bytes of a type we write next to it between parenthesis. It is also capable of calculating the size of complex expressions, but we will see that later on. For now, just remember that `sizeof(int)` is equal to the size in bytes of an integer. As you can see, I simply multiply the size of the integer by the number os positions that I know that are unique. The next loop, simply, copies from the preliminary solution to the definitive. Finally, we return the pointer allocated with `malloc`.

In `main` function we simply declare an array with several random numbers with repetitions, we call the function on them and show the result on the screen. You'd see that it works as it is intended. Note the use we make of the operator ampersand to pass to the function `erase_reps` the pointer to a normal variable.

As a note, I have been using and mixing vector and array in this section and I didn't do it casually: a vector is a contiguous memory block that stores data, which is susceptible or growing and shrinking and that, consequently has been reserved dinamically. An array is that data type I explained to you on its own section. They're alike, and they share many characteristics, as a matter of fact, but they're not exactly the same.

And now we're talking about vectors and arrays, there is a fundamental diference between vectors or pointers and arrays. We **can ask for the size of an array** but not of vectors. And, if we can know the size of arrays, why have we been using a literal value? Because I didn't want to show you this until we could compare arrays and vectors. Before, I told you that `sizeof` gives us the size in bytes of a data type or **an expression**. The size in bytes of an array is what one would expect: if it contains ten integers of four bytes, the size would be 40. But the size of a pointer **is always the same**. Furthermore, the size of a pointer to any datatype is always the same. Let's see how to use `sizeof` to get the size of an array.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int array[] = { 12,42,53,85,45,54,11,26,21,13 };
7      int array_size = sizeof array / sizeof array[0];
8
9      for(int ii = 0; ii < array_size; ++ii){
10         printf("%d\n", array[ii]);
11     }
12
13 }
```

Program 57: Difference of `sizeof` between pointers and arrays

The fact that `sizeof` is a operator and not a function comes into play here. If you look at line 7, we use the operator `sizeof` to get the size of the array and divide it by the size of the first element. Mind you that `sizeof` doesn't need to know what is **inside** the first position, just the type of the expression (`array[0]`) which would be an `int`. You could use `sizeof` like this over memory that is not accessible and it won't give you any problem, because it does not read the content but only evaluates the type of the expression. This way, we can calculate easily the number of positions. Here I have assigned it to a variable so you can see it more clearly, but you could have put this expression directly in the `for` loop. Well, knowing this, maybe you are wondering why we always passed the lenght of array to functions if we could know it. The answer is that we do not know it, because you must remember that, when a function gets an array passed as argument, this decays to be a mere pointer. The good side of vectors, nonetheless, is that since we reserve them using their size, you can assign it to a variable before doing the allocation to use it later several times.

And I still have another trick to teach you about this operator, and it is that allows us to write the calls to `malloc` in a way that favours some changes in the code. Imagine this call to malloc: `int *vector = malloc(length * sizeof(int));`. It is like the one we did before, but it presents a little problem, if we change the type of the vector we must be very careful to also change the type that is inside the `malloc`, because, if not, we would be allocating less memory than we want. Nevertheless: let's remember that `sizeof` allows us to calculate the size of expressions, so we can change the call to something like this: `int *vector = malloc(length * sizeof *vector);`. Pay attention to it, if `vector` is a pointer to `int`, `*vector` would be the first element, an integer, and `sizeof` will hence give us the size of an integer (4). The advantage of this style of call is that, if we change the type of the vector, we do not have to remember changing anything inside `malloc`.

8.5. Pointer composition

Now you already know the basic mechanisms of pointers, we can see some of the examples of more complex structures made with them. For example, one of the most interesting cases that you may encounter when programming is the pointer to pointer to a given data type. It is equivalent to a bidimensional array, but with dynamic memory allocation. Let's write a program in C that creates this structure, I encourage you to compare it with the program 34: Ejemplo de uso de array bidimensional, in which we saw the creation and use of a bidimensional array.

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6       int rows = 10;
7       int columns = 5;
8       int** matrix = malloc(rows * sizeof(*matrix));
9
10      for (int ii = 0; ii < rows; ++ii) {
11          matrix[ii] = malloc(columns * sizeof(*matrix[ii]));
12          for (int jj = 0; jj < columns; ++jj) {
13              matrix[ii][jj] = ii * columns + jj + 1;
14          }
15      }
16
17      for (int ii = 0; ii < rows; ++ii) {
18          for (int jj = 0; jj < columns; ++jj) {
19              printf("%2d\t", matrix[ii][jj]);
20          }
21          printf("\n");
22      }
23
24      for (int ii = 0; ii < rows; ++ii) {
25          free(matrix[ii]);
26      }
27      free(matrix);
28  }
```

Program 58: Reserva, uso y liberación de un vector de vectores

In line 8 of the example program you can see that we declare a pointer with two asterisks, this is because it is a pointer to pointer to integer. The pointers can be chained indefinitely and, actually, there is no reason for it not to be that way. If a pointer is a memory address, nothing avoids me from making this to point to a place in memory where another address is, and so on and so forth. Also, you can see that we allocate memory for `rows` pointers to integer. The rule to understand pointers is that the number of asterisks in the declaration is compensated by the number of asterisks used to dereferencing, in this way, if we declare `matrix` as an `int**`, `*matrix` is an `int*` (two asterisks in declaration minus one in dereferencing). Look at the simmetry, or simply count the asterisks.

In line 10 we start a `for` that allocated the memory for each row of `matrix`, later, once we have allocated the memory, we fill those positions with a value in the loop in line 12. Here simply we're making each position to be equal to its overall position (starting in one, for a change). As you can see, we're using square brackets to index this double pointer, with brackets it's the same that with asterisks, if we have `matrix` that is an `int**`, doing `matrix[ii][jj]` we are getting an `int`. The two nested loops simply print the matrix.

Finally, we must free the matrix, note that this is also simmetrical with the allocating process, if firstly we did a `malloc` of `row` pointers and then each one of those was allocated with a malloc of `columns` integers, here we liberate the pointers in the reverse order, firstly we free each row and later the matrix itself.

Since this can be confusing the first time you see it, I am going to draw the memory map of this situation, so you have a visual image of what is happening. Pointers are a very abstract concept, so don't worry if you do not understand them at the first glance.

0x0000000000000000	↕
matrix	0x7fffba2522a0
	↕
matrix[0]	0x7fffba252710
matrix[1]	0x7fffba252730
matrix[2]	0x7fffba252750
matrix[3]	0x7fffba252770
matrix[4]	0x7fffba252790
matrix[5]	0x7fffba2527b0
matrix[6]	0x7fffba2527d0
matrix[7]	0x7fffba2527f0
matrix[8]	0x7fffba252810
matrix[9]	0x7fffba252830
	↕
matrix[0][0]	1
matrix[0][1]	2
matrix[0][2]	3
matrix[0][3]	4
matrix[0][4]	5
	↕
matrix[1][0]	11
matrix[1][1]	12
matrix[1][2]	13
matrix[1][3]	14
matrix[1][4]	15
	↕
matrix[2][0]	21
matrix[2][1]	22
matrix[2][2]	23
matrix[2][3]	24
	↕
0xFFFFFFFFFFFFFFFF	

Figure 9: Mapa de memoria de un vector de vectores (doble puntero)

In the figure I present the memory map, on the left there are the names those locations have in the program, and in the rectangle I write their contents. Arrows on the right side of the image represent the references of the pointers to those memory addresses. Colors simply allow you to follow the different arrows more easily. Well, if you look at where I put the tag `matrix`, you'd see it contains a memory address, this address references to a **vector of pointers**, that is, `rows` pointers together. Each one of those pointers, to a contigous memory region in which the is an antire row saved. I have only represented the first three rows, because otherwise the image would be too big.

Now I am going to draw the figure on how would the map be in the case of a double array, I don't include the code bcause it would be simply: `int matrix[10][5]`.
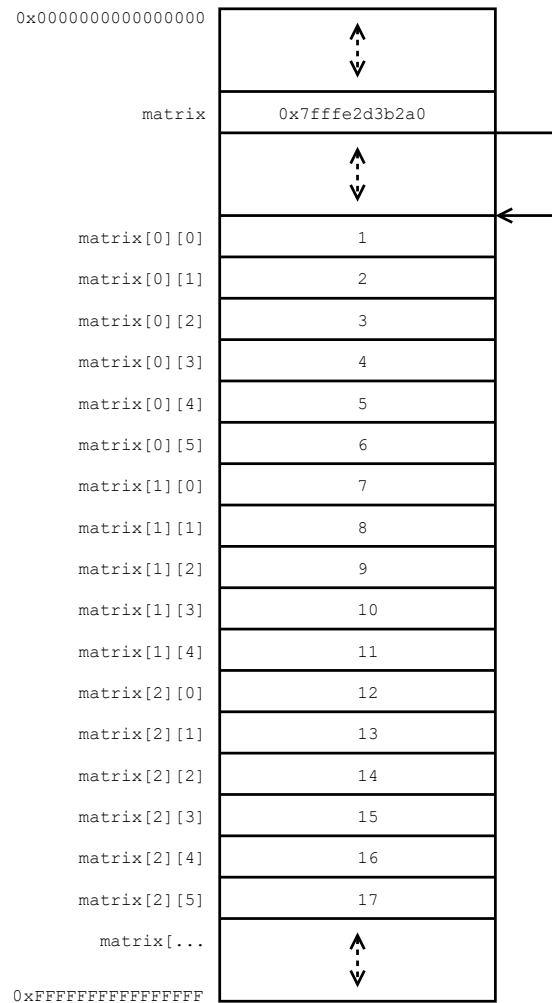
Figure 10: Memory map of a double array

As you can see, even when the array have been declared with two dimensions, there is not dereferenciation. That is: there is no moment in which you follow a second pointer. How is this possible? If you look to the map, you'd see the array is stored in a contigous memory region. This meand that C only needs to adquiere the start address of the array and, later, add what the indexes inside the square brackets tell you. It's here where a problem arises when we try to pass a bidimensional array to a function. When the array reached the function, C doesn't know if that pointer is a bidimensional array or a vector of vectors, that is why, if you passed this array to a function that receives a double pointer (int**), when performing, for example matrix[1][2] what it would try to do is accessing it like it's a pointer, and would do: *((*(matrix + 1))+2). That is, it would firstly add one to the base address (remember pointer arithmetic) and then **it would interpret the content as a pointer** to another vector and it would try to add 2 to that address to dereference it. The problem is that *(matrix+1) **is not** a pointer, it is directly a number.

C can do this because, in the same that I explained you how the operator sizeof works, we can see the C knows the size of an array as long as this does not decay to pointer, that is, you can know the size of an array in any scope inner to the one it was declared in.

The logic conclusion of what we have just learnt is that maximun number of dimensions of an array that any function can get is one, because it is the one that behaves as a pointer without problems. The fact that an array, by being declared in a unique order, is contiguous, makes arrays able to be accessed always a one dimensional structures. For example, in the next code we declare and fill up an array of two dimensions and, nevertheless, we can access it like if it had just one dimension.

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

int main(void)
{
    int rows = 10;
    int columns = 5;
    int matrix[10][5];

    printf("matrix = %p\n", matrix);

    for (int ii = 0; ii < rows; ++ii) {
        for (int jj = 0; jj < columns; ++jj) {
            matrix[ii][jj] = ii * columns + jj + 1;
        }
    }

    for (int ii = 0; ii < rows * columns; ++ii) {
        printf("%d\n", (*matrix)[ii]);
    }

}
```

Program 59: Using a bidimensional array like a one-dimension structure

As you can see, once we have reached the contiguous memory region ((*matrix)), we only have to iterate over it like if it were a one dimension array.

8.6. Section exercises

Ex. 10: Fill up this table with different numeric bases:

Decimal	Binary	Hexadecimal
73		
	00100110	
		0x12F
128		

Ex. 11: Return to the code of the 9th exercise and reproduce the content of the stack in all the code blocks of the program. Use as code the solution I propose in the solution section.

Ex. 12: Write a program that creates a pointer of three levels of type `int`, allocates memory for it correctly and fills it up with correlative values **starting in one** and later on it prints it in an understandable way. Finally, free it also in a way that the is no left unfreed memory at the end of the program.

Ex. 13: Using the last program as base, create two functions, one to create a tridimensional matrix with dynamic memory given its three dimensions and another one to free it.

9.   Type modifiers: const and sign

Up until now we have only used basic types, but you can add **modifiers** to those types, which are qualities that create a slightly different type based on the original one. The first and most important is the `const` modifier. This allows us to indicate that the value of a variable is **read only**, that is, once we give it value, and we must do so when declaring it, we couldn't modify it. This is very useful to make sure we do not introduce errors in the code when we use data that shouldn't be changed.

For example, imagine a program that uses the number π. Let's say to calculate the area of a circle, we would need to write `surface = PI * r * r`, but imagine we make an error and write `surface = PI *= r * r`, this error would be difficult to catch because the first time the surface would be correctly calculated, but `PI` would have a different value because we have unwillingly used the operator `*=`, and all the later calculations would be erroneous. If we declare `PI` as a constant, the compiler could notify us about those errors.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  const double PI = 3.141592;
5
6  double surface_circle(double radius) {
7      return PI * radius * radius;
8  }
9
10 double perimeter(double radius) {
11     return 2 * PI * radius;
12 }
13
14 int main(void)
15 {
16     double r = 3.5;
17     printf("A circle with a radious of %.3f has a perimeter of %.2f and
              an area of %.2f\n", r, perimeter(r), surface_circle(r));
18 }
```

Program 60: Uso de una constante numérica

In line four you can see how we declare `PI` as a constant of type `double`. You can see, also, we are declaring it as a global variables. If you remember when I explained the scope and what global variables are, I told you they would be useful once we had functions. Here you can see one of the most common uses of them, when you define universal contants, as π, e or the gravitational constant G. But coming back to the constant, if you try to write an instruction that modifies the constant, the compiler would throw an error like this one:

```
$  gcc -o main.exe main.c
main.c: In function 'main':
main.c:17:8: error: assignment of read-only variable 'PI'
   17 |     PI = 1.1;
      |        ^
```

But the `const` modifier is applied in other place, it is used in the declaration of arguments of functions to indicate that those cannot be modified. Again, arguments of functions are copies of their values, so, knowing this, it wouldn't make any sense to say we can't modify them, the core matter about this is that the `const` modifier is applied to pointers, indicating that their content can't be modified, and this is where it is tremendously useful when joined with functions. For example, let's go back to the function that calculates the distance between two points using pointers, since we do not want the structure to be modified by the function, we can do this:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  struct point_s {
6      double x; double y;
7  };
8
9  double distance(const struct point_s* a, const struct point_s* b) {
10     double res = 0.0;
11     struct point_s origin = { .x = 0.0 , .y = 0.0 };
12     if (NULL == a) {
13         a = &origin;
14     }
15     if (NULL == b) {
16         b = &origin;
17     }
18     double diff_x = a->x - b->x;
19     double diff_y = a->y - b->y;
20     res = sqrt(diff_x * diff_x + diff_y * diff_y);
21     return res;
22  }
23
24  int main(void)
25  {
26      struct point_s a = { .x = 3 , .y = 4 };
27      double d = distance(&a, NULL);
28      printf(" Distance : %f\n", d);
29
30  }
```

Program 61: Uso de punteros constantes como argumentos de función

If you look at the program, the only difference is that in the declaration we put `const` before the data type. This avoids that we modify the content of that pointer inside the function. If you try to perform for example: `a->x++;` the compiler will throw an error. This is very useful for the programmer that uses the function if he hasn't written it, because that declaration is a way to tell him that the function does not modify the data at all. When we reach the point of writing our programs in different source files, we will see this in more depth.

Also, the modifier `const` adds the concept of **const correctness**, this concept means the programmer needs to respect the quality of constance of variables and arguments. This means that you must define your functions carefully, indicating everything you can as constant. For example, in the case of the function that calculated the distance between two points, both arguments must be constant. But there is more, the function **can return a constant**. This is useful when you create structs that you do not want the user to modify, but only with the functions you provide for that.

We will talk about this further later on, but I am going to set a basic example: imagine a struct that saves data of a person. In this struct we would have several pointers to `char`: the name, the first surname and the second surname (I wrote this example for Spanish names, which have two surnames). What we will do is to create a function that receives three texts, will allocate the necessary memory and we are going to create functions that replace those vectors when the user of the functions needs to do so. Let's go:

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct person_s {
6      char* name;
7      char* last_name_1;
8      char* last_name_2;
9  };
10
11 struct person_s person_create(const char* name,
12                               const char* last_name_1,
13                               const char* last_name_2) {
14     struct person_s res;
15
16     res.name        = malloc(strlen(name) + 1);
17     res.last_name_1 = malloc(strlen(last_name_1) + 1);
18
19     for (int ii = 0; ii < strlen(name) + 1; ++ii) {
20         res.name[ii] = name[ii];
21     }
22
23     for (int ii = 0; ii < strlen(last_name_1) + 1; ++ii) {
24         res.last_name_1[ii] = last_name_1[ii];
25     }
26
27     if (NULL != last_name_2) {
28         res.last_name_2 = malloc(strlen(last_name_2) + 1);
29         for (int ii = 0; ii < strlen(last_name_2) + 1; ++ii) {
30             res.last_name_2[ii] = last_name_2[ii];
31         }
32     }
33     else {
34         res.last_name_2 = NULL;
35     }
36     return res;
37 }
38
39 void person_set_name(struct person_s* person, const char* name){
40     free(person->name);
41     person->name = malloc(strlen(name)+1);
42     for(int ii = 0; ii < strlen(name) + 1; ++ii){
43         person->name[ii] = name[ii];
44     }
45
46 }
47
48 void destroy_person(struct person_s *person){
49     free(person->name);
50     free(person->last_name_1);
51     free(person->last_name_2);
52 }
```

Program 62: Struct with const pointers – Managing functions

```
1  const char* person_get_name(const struct person_s* person) {
2      return person->name;
3  }
4
5  const char* person_get_last_name_1(const struct person_s* person) {
6      return person->last_name_1;
7  }
8
9  const char* person_get_last_name_2(const struct person_s* person) {
10     if (NULL == person->last_name_2) {
11         return "";
12     }
13     else {
14         return person->last_name_2;
15     }
16 }
```

Program 63: Struct with const pointers – Functions to retrieve information

```
1  int main(void)
2  {
3      struct person_s myself = person_create("Francisco", "Rodríguez", "
           Melgar");
4
5      printf("Esta persona es: %s %s %s\n", person_get_name(&myself),
6              person_get_last_name_1(&myself),
7              person_get_last_name_2(&myself));
8
9      person_set_name(&myself, "José");
10
11     printf("Esta persona es: %s %s %s\n", person_get_name(&myself),
12             person_get_last_name_1(&myself),
13             person_get_last_name_2(&myself));
14
15     destroy_person(&myself);
16 }
```

Program 64: Struct with const pointers – `main` function

In this code you can see how we "hide" the user how we manage these pointers. To avoid him from chaging its content without using our functions, the functions that return the pointers to be able to use them, for example to print them, return constant pointers. If you tried to do, for example: `person_get_name(&myself)[3] = 'a'` the compiler would throw and error. This is a tool to avoid the user of the struct from forgetting to free the memory when replacing a text by other.

If you're paying attention, you may have noticed that all of this is a little bit useless when the use can simply write something like: `myself.name[0] = 'a'` and you'd be right. In later sections we would see ways to avoid this. But, even with this problem, doing this is a good way to save work to the user of the struct.

Another modifier I want to present to you is the sign, or better said, the absence of sign. The keywork `unsigned` allows us to declare variables and arguments of the same types of the basic types, but that can only contain positive integers. If you do not see at first sight why this is useful, this resides in that making a type unsigned you get a range doubled in the possitive side. If a `char` has a range of $[-127, 128]$, an unsigned char has a range of $[0, 255]$. Also, this allows to add meaning to your variables.

For example, a variable that stores the size of something, for example an array, or a vector, shouldn't have a sign, because it can never be negative. A variable that stores a month, shouldn't either, for instance. In the case of sizes of vectors it is important to make the variable that stores their size unsigned, because this allows us to get vectors and arrays of double the size. There are not types without sign for floating point numbers (`float` and `double`). Let's see an example on how to use the modifier `unsigned` to declare variables, arguments and return types.

```c
#include <stdio.h>

unsigned int factorial(unsigned int n) {
    unsigned int res = 1;
    for (unsigned int ii = 0; ii < n; ++ii) {
        res *= n - ii;
    }
    return res;
}

int main(void)
{
    unsigned int number = 10;
    printf("%d! =%d\n", number, factorial(number));
}
```

Program 65: Use of unsigned types

I know it looks a bit cumbersome to write unsigned each time, later on we will see how to solve this.

9.1. Exercises of the section

The most appropiate exercise is for you to revise all the exercises we have done and rewrite the code having into account the const quality and the sign.

10.   Communicating your program

At last, we're reaching the part of the manual where you can communicate your program with things outside it, up until now, all the data we have introduced in the program are written as literals. This is very impractical, generally, a program will read, either from the terminal or from a concrete file, the data it is going to use. There are three sources of external information for a program we're going to see in this manual (there are many more):

1. Arguments from the command line.

2. Input from the terminal.

3. Files

The first thing is something you do not know yet, but it's going to be very useful. Up until now, `main` function wasn't receiving any argument, but how can it do so? If `main` is the function that only acts as the entry point for our program, who can call it with arguments? Basically these arguments come from the command line we have executed our program with. To be able to access them inside the program, we must declare the function `main` in this new way:

```
1  int main(int argc, const char** argv) {
2  // ...
```
Program 66: Declaration of a `main` function that receives arguments

Of these two arguments of the function, the first one is the number of arguments the program has received, and the second is a vector of vectors of `char` that is sent to us as a two-level pointer. The arguments that a program receives come in text format so, if they're numbers, you must use some functions to turn them into those data types. Let's make a program that receives an undeterminated number of arguments and prints them, each one in a new line.

```
1  #include <stdio.h>
2
3  int main(int argc, char const *argv[])
4  {
5      for (int ii = 0; ii < argc; ++ii) {
6          printf("%s\n", argv[ii]);
7      }
8  }
```
Program 67: Ussage of the arguments or a program

Maybe you're wondering how can we pass those arguments to the program, after the path of the executable, you write all the arguments separated by spaces, for example:

```
$./main.exe argument1 argument2 argument3
```

If you compile the program and execute it with those arguments, it will print this:

```
./main.exe
argument1
argument2
argument3
```

And yes, as you can see, the first argument is the command you called the program with, and this is important because, as you may deduct, this means that your programs always receives at least one. This first argument changes according to the order you executed your program with, for example, if instead of with that path you'd execute it with an absolute path, you'd get the following.

```
$ /home/usuario/test/project/main.exe arg1 arg2 arg3
/home/usuario/test/project/main.exe
arg1
arg2
arg3
```

As a last note, if the space is the character that separates the arguments, how do we write arguments with spaces in them? Simply surround the argument between straight quotes (like C strings). Let's see an example.

```
$ ./main.exe "This is a double quote: \"" \"\"\" "Sentence with spaces"
./main.exe
This is a double quote: "
"""
Sentence with spaces
```

If we need to read a number, we must use, as we said before, a function that turns the text into a number. The basic function to do this is `atoi` (from ASCII to integer). For example, let's create a program that receives a series of numbers and adds them up.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char const* argv[])
5  {
6
7      int total = 0;
8
9      for (int ii = 1; ii < argc; ++ii) {
10         total += atoi(argv[ii]);
11     }
12
13     printf("The addition of the arguments is: %d\n", total);
14 }
```

Program 68: Program that adds its arguments

Mind that we start adding from the second argument (position one in the vector) because the first one is not a number. This leads me to warn you that `atoi` is a very basic function and that if you pass something to it that is not a number, it will return nonsense values, many times zero. So be careful about this. Anyway, in other sections we will see ways to manipulate text strings that are more complex and would allow us to check it. It would be an interesting exercise that you made a program that checks if a text string is a number or not. Finally, there is also `atof` that does the same, but for floating point numbers.

Another option is to allow the user to write things into the program once it has started. For example, we could write a program that is a 2.0 version of our first *Hello, World*. This version would firstly ask the name of the user and then say hello to him personally. You can use the function `scanf` to do so. This function is the twin of `printf` because it behaves in the same way, you specify a format and pass the variable you need to it where it will store the data. For example, let's make this *Hello World 2.0*

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define NAME_LENGTH ((size_t)1024)
5
6  int main(int argc, char const* argv[])
7  {
8
9      char name[NAME_LENGTH] = "";
10     printf("Hello, what's your name?\n");
11     scanf("%s", name);
12     printf("Nice to meet you, %s.\n", name);
13 }
```

Program 69: `scanf` basic example

As you can see, it's easy, but you must be aware that if you read basic types you must pass the pointers to the variable your want to store them in to `scanf`. With this `char` pointer it's less evident. Also, unless you call other functions to change how the terminal behaves, `scanf` will only read up until the first whitespace character. That is, until the first space. This means that if you want to read several words at the same time you must call `scanf` several times or call it with several format specifiers. Let's see a bit more complicated example.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define NAME_LENGTH ((size_t)1024)
5
6  int main(int argc, char const* argv[])
7  {
8
9      char name[NAME_LENGTH] = "";
10     int age = 0;
11     double height = 0.0;
12     printf("Hello, I want to get to know you, tell me your name, age
           and how tall are you in meters.\n");
13     scanf("%s %d %lf", name, &age, &height);
14     printf("Nice to meet you, %s. So you're %d years old and are %f m
           tall.\n", name, age, height);
15 }
```

Program 70: `scanf` advanced example

As we said, `scanf` stops reading in the first space, so your can write each data and hit enter or write the three answers separated by spaces and hit enter just once. Also, note that, as you want to read a `double` we must use the specifier `%lf`, the compiler would warn us this is not right if we used `%f`. This function **blocks** the program until is receives **all** the arguments we're asking for with the specifiers. Also, what is written in the terminal and does not get used (is read with `scanf`) stays there pending to be used. That is, if instead of writing our example with just one call to `scanf` we wrote three calls, each one with one specifier and variable, the effect would be the same. You could still write the three things separated by spaces and hit enter. In the same way that `atoi`, if the specifiers expects a number and the input is not a number, the function will fail throwing values with little to no sense.

Finally, the files. Programs can delete, create, write and read from files. To do this, there are several ways, some more simple than others, because some are more standard and other depend on the operating system. In opposition to the other methods, files have a more complex life cycle. Life cycle in this context refers to the description of when something starts to exist, exists and ceases to do so. For example, the life cycle of a dynamically reserved memory block is from the moment of their allocation to the moment it's freed, the one of an array goes from the moment you go into the block of code it's declared to the moment the program stops executing in that block.

Files are a concept that, again, the operating system manages, therefore we have to use functions to open, write, read from them and, finally, close them. If you have been paying attention, you may have seen the parallelism between this and memory allocation. The function to open a file is `fopen`, let's see its declaration.

```
1  FILE *fopen(const char *pathname, const char *mode);
```
Program 71: `fopen` function declaration

It returns a pointer to a type called `FILE`, this type is **opaque**, this a word that it is used in computer science to say that you can't know what is inside, that is, it is the system who manages it and you only interact with it calling the functions it provides you. The arguments this functions receives are two `char` pointers. The first one is the path to the file. This path takes as origin **the working directory** of the terminal where we executed the program. That is, if you pass to the function the path `./file.txt`, it will look for it (or create it, depending on how the function was called), in your working directory, no in the one the program is in, unless they're the same.

The second argument is interesting, it is, as its name says, the mode in which we're opening the mode. In this argument you tend to write a literal that contains a series of letters, these letters are the attributes of the way you're going to open the file. Let's see which options we have.

1. `r`: Open the file just to read it, the reader head (more on this later) is at the start of the file. If the file does not exist, an error occurs and the function returns `NULL`.

2. `r+`: Same as before, but it allows to write too.

3. `w`: **Empties** the file or creates it if it doesn't exist. The reading head is at the start, logically.

4. `w+`: Like the last one, but allows to read too.

5. `a`: Open the file to write in it, but **it does not empty it**. It creates it if it doesn't exist, the head start at the end of the file, to **append** data to it.

6. `a+`: The file is created if it does not exist, data will be written to the end of the file. It also allows reading.

In the description of the options I talk about something called the reading or writing head, also called read/write pointer. It is not a C pointer or anything like that. It is a concept inherent to files. The read/write head is an abstraction of a physical device that is in a point of the file and writes or reads to it. Files are just a stream of bytes, and this head would write or read from the position it is in, and advance as many bytes as it read or wrote. The methaphor that was used was a video tape. If you put a tape in a video player, the head would start at the begining of the tape, if you watched let's say ten minutes of it, you could pause it, go do something and come back, it will be where you left it. You could rewind it, fast forward it, etc. Also, playing the video makes the head move forwards, that is: reading makes you rewind if you want to read the same thing again. All this being said, the read/write header is unique **per file and process**, this means that two programs can be reading or writing from the same file at the same moment but in different positions. When you reach the end of the file you can't read more (logically) but if you're writing, the file would grow seamlessly.

Now you already know how to open a file, let's see how to write or read to and from it. The functions that are used to do this are these two: `fwrite` and `fread`

```
1  size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```
Program 72: `fread` function signature

```
1  size_t fwrite(const void *pt, size_t size, size_t nmemb, FILE *stream);
```
Program 73: `fwrite` function signature

The two functions behave more or less the same, they receive a pointer, a size, another number a finally the pointer to the struct that symbolizes the file. The first pointer is the input to write and the output to read. It is where we will put our data, or the memory so the function can write the content of the file in it. It is a `char` pointer, which is the kind of pointer used when you want to manage a generic stream of bytes. Both functions are written with the idea of managing a number of items of a certain size in bytes, that's why we have `size` and `nmemb`. The first one is the size of the items we're writing or reading and the second the number of said items we're reading or writing. The function returns the number of **items** written or read, not the number of bytes, mind this when using the functions. More on this in the examples later on.

Finally, after doing what you want with the files you have opened, you must close it. Closing the files makes all the changes you have done to it to be written with the underlying storage device. This is important, if you forget to close a file you may see the changes you've done in your program are not in the real file. The function to close files is called `fclose` and its signature is like this:

```
1  int fclose(FILE *stream);
```
Program 74: `fclose` signature

As you can see, it only receives the pointer to the file we can to close.

Now we have shown the functions that are needed, let's an example on how to use files. A very usual program is one that copies a files from one directory to another, like the command `cp`, which is the one used in Linux to do so. Let's make a program that receives two arguments, the first one would be the file we want to copy and the second the folder we want to copy it to. The Linux command allows the use to specify a folder as destiny, but to make things simpler we will make this program to ask for a full path (name of the destiny file) included.

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

int main(int argc, char const* argv[])
{

    FILE *origin_file  = NULL;
    FILE *destiny_file = NULL;
    char byte          = 0;

    if (argc < 3) {
        printf("Use: main.exe <origin> <destiny>\n");
        return EXIT_FAILURE;
    }

    origin_file = fopen(argv[1], "r");

    if (NULL == origin_file) {
        printf("ERROR: The origin file doesn't exist.\n");
        return EXIT_FAILURE;
    }

    destiny_file = fopen(argv[2], "w");

    if (NULL == destiny_file) {
        printf("ERROR: The destiny file doesn't exist.\n");
        fclose(origin_file);
        return EXIT_FAILURE;
    }

    while (0 != fread(&byte, sizeof(char), 1, origin_file)) {
        fwrite(&byte, sizeof(char), 1, destiny_file);
    }

    fclose(origin_file);
    fclose(destiny_file);
    return EXIT_SUCCESS;
}
```

Program 75: Example of basic file management

As you can see, we have declared two variables of the pointer type to FILE, we initialize them to NULL and start the program. When you receive arguments it is adviceable to parse them first, because if they are not correct, there is no purpose in continuing with the program. After that, we try to open the origin file, note that we open it just to read it and without emptying it, logically. If that call fails, the value it return would be NULL and we check this is not the case before continuing. We do the same with the destiny file, but this one is opened with the option r that will empty the file if it exists and create it if it does not. Notice that, in case of error, we need to close the first file we opened before exiting the program (returning from function main ends the program). Finally, we use a loop to read byte by byte the origin file and write it in the destiny. As you can see, the loop condition is that it will keep looping as long as the reading function returns anything that is not zero. This is like so because both fwrite and fread return the number of items they have written or read. In this case, when fread returns zero it is there is no more to read.

I am going to take some time explaining the return value of the functions, as we said before, we pass to both `fread` and `fwrite` a size and a number of items we want to read or write. This would allow us for example to write or read a certain number of structs whose size we could know with `sizeof`. The return value is the number os **items** read, not the number or bytes. When writing a stream of bytes directly, simply indicate that the size is one and `nmemb` is the numbers of bytes.

You may be thinking that writing a program like this that reads one byte at a time is a bit inefficient, and you're right. Each call to a function that manages things that are handled by the operating system is relatively costly, so it is intelligent to minimize them. In this case, we could read all the first file in memory and then close it and write it in the second, in just one order. The problem with this is that if you tried to cpy a 12 GB file it's probable that you fill all the memory in the computer, in which case it simply hangs or closes the program. To avoid both extremes, what is done often is to read a block of a sensible size (for example 100 MB) and write it, and repeat until the process is done.

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

int main(int argc, char const* argv[])
{

    const size_t BLOCK_SIZE   = (size_t)(1024 * 1024);
    FILE*         origin_file  = NULL;
    FILE*         destiny_file = NULL;
    char*         buffer       = malloc(BLOCK_SIZE);
    size_t        bytes_read   = 0;

    if (argc < 3) {
        printf("Use: main.exe <origin> <destiny>\n");
        return EXIT_FAILURE;
    }

    origin_file = fopen(argv[1], "r");

    if (NULL == origin_file) {
        printf("ERROR: The origin file does not exist.\n");
        return EXIT_FAILURE;
    }

    destiny_file = fopen(argv[2], "w");

    if (NULL == destiny_file) {
        printf("ERROR: The destiny file does not exist.\n");
        fclose(origin_file);
        return EXIT_FAILURE;
    }

    while (0 != (bytes_read =
                    fread(buffer, sizeof(char), BLOCK_SIZE,
                        origin_file))) {
        fwrite(buffer, sizeof(char), bytes_read, destiny_file);
    }

    fclose(origin_file);
    fclose(destiny_file);
    free(buffer);
    return EXIT_SUCCESS;
}
```

Program 76: Example of file copying with a buffer

The only change is that instead of a single `char` we declare a vector of them with the size defined in a const variable and, later on, in the copy loop, instead of copying always one byte, we try to read the block size and, when writing it, we use the value returned from the read operation. I want to take a moment in this line because it's a bit different from what we have seen up until now. An assignment is an expression with a value. This is what allows us to do things like: `a = b = c`. Thanks to this property, you can compare the result of an assignment to another value. In this case we could save writing the comparison because putting a number in an if is the same that checking it is not zero.

This is a basic use of reading and writing, but anochet thing that is done often is **move the read/write pointer** without reading or writing. For example, imagine that we want to print a file in reverse order. We are in the same predicament as before, we could read all the file and then invert it, but it is a problem again if the file is too big. We could do the same we did before, read the file piece by piece, invert the pieces and then put them all together, also in a reverse order. This is complicated, thanks to the fact we can move the read/write pointer, we can read directly the blocks in reverse order.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define BLOCK_SIZE ((long) 100)
6
7  void invert_bytes(char* stream, int length)
8  {
9      for (int ii = 0; ii < length / 2; ++ii) {
10         char temp = stream[ii];
11         stream[ii] = stream[length - 1 - ii];
12         stream[length - 1 - ii] = temp;
13     }
14 }
15 int main(int argc, char** argv)
16 {
17     long  current_pos = 0;
18     FILE* file = NULL;
19     if (argc != 2) {
20         printf("Uso del comando: main.exe <archivo>\n");
21         return EXIT_FAILURE;
22     }
23     file = fopen(argv[1], "r");
24
25     if (NULL == file) {
26         return EXIT_FAILURE;
27     }
28
29     fseek(file, 0, SEEK_END);
30     current_pos = ftell(file);
31     while (current_pos != 0) {
32         char block[BLOCK_SIZE + 1] = {};
33         long next_pos = 0;
34         long block_size = 0;
35         if (current_pos - BLOCK_SIZE < 0) {
36             next_pos = 0;
37             block_size = current_pos;
38         }
39         else {
40             next_pos = current_pos - BLOCK_SIZE;
41             block_size = BLOCK_SIZE;
42         }
43         fseek(file, next_pos, SEEK_SET);
44         fread(block, 1, block_size, file);
45         fseek(file, -block_size, SEEK_CUR);
46         invert_bytes(block, block_size);
47         printf("%s", block);
48         current_pos = ftell(file);
49     }
50     fclose(file);
51     printf("\n");
52 }
```

Program 77: Example of use of functions to move the read/write pointer

The program is a bit complicated, but, as always, we will explain it little by little. The first thing we do is to define a function that can reverse the bytes of a vector. It is not the main matter in this example, but remember the algorithm because it is a classic. What we want to do starts right after, it is a bit complex, but I am interested in you seeing what the functions that move the head do. Up until line 27 the only thing we do is something you're already used to see: we process the arguments and open the file, checking everything has gone well. After that we use the function `fseek` that allows us to **move the read/write pointer**, concretely, we move it to the end of the file. Let's see how it work.

This function allows us to move the head to a point in the file with an offset. There are three points you can use as reference for the offset.

1. `SEEK_SET`: The start of the file. `fseek(100, SEEK_SET);` would set the head to be in the 100$^{\text{th}}$ byte of the file. When using this point, you cannot use negative offsets, logically.

2. `SEEK_CUR`: It is the current position, you can use negative of positive displacements For example: `fseek(100, SEEK_CUR);` could be used in a loop to read the bytes 100$^{\text{th}}$, the 200$^{\text{th}}$, the 300$^{\text{th}}$...

3. `SEEK_END`: It is the end of the file, for example if you wanted to go to the third byte from the end you'd perform: `fseek(-3, SEEK_END);`

In line 29 what we do is going directly to the end of the file. After that, we use another function called `ftell`. This functions give us the current position in the file as a number. That is how we know where we are. After that we check if there is a whole block of data left. If we can, we use that block size, if not, we use what we can. After that, we displace the head to the position we have calculated, and read the amount of bytes we calculated. We read and **displace the pointer back again** as many bytes as we read. After that we call the function that inverts the bytes we have read. After that, we update the variable that tell us where we are. Once we arrive to the first position of the file, we know we have finished. You must be careful, `fseek` only works when it is possible to move the pointer to the place you tell it to. When it is not possible, it doesn't move the pointer and gives an error (returns -1). Finally, we close the file and print a new line so the prompt shows in the next line.

We haven't used it in the example, but you can **delete** files. To do so we use the function `remove`, whose signature is:

```
1  int remove(const char *pathname);
```

Program 78: `remove` function signature

As you can see, it simply receives a path. Let's see an example of a simple program that makes use of it: one that receives as an argument a path and deletes it.

```c
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int main(int argc, char const *argv[]) {
 5      if (argc != 2) {
 6          printf("Usage: ./main <path to the file>");
 7      }
 8
 9      int error = remove(argv[1]);
10
11      if (error == 0) {
12          return EXIT_SUCCESS;
13      } else {
14          return EXIT_FAILURE;
15      }
16  }
```

Program 79: Example of a program that uses `remove`

The program is very simple, I call remove with the first argument (after the name of the program) and check is it has gone well.

10.1. Exercises of the section

Ex. 14: Write a program that receives a set of number as arguments and prints their decomposition in prime factors, for example, if the arguments were: 10, 8, 55 and 103, the result would be:

```
Factors of
10: 2, 5
8: 2, 2, 2
55: 11, 5
103: 103
```

It is recommended to make error control, checking the arguments are numbers before using them, etc. Also, the order of the factors is not important.

Ex. 15: Write a program that reads **from terminal** a series of words and that only stops reading when you introduce "!!" as a word. After that, it must print said words in a random order. The function `rand` returns a random number between zero and the biggest possitve integer, you need it to generate random numbers. If you want the random numbers to be different from one execution to the other, simply put this line: `srand(time(NULL));` at the begining of your program. If you decide to do this too, you need to add the line `#include<time.h>` just after the `#include<stdio.h>` line.

Ex. 16: Make a function that reads two files and **swaps** their content, write said program in such a way that there is no need to store any of the files in memory completely. To do this you may follow these steps.

1. Copy the contents of the first file to an auxiliary file in `/tmp/`.

2. Close the file you have copied.

3. Open it again with a mode that deletes the contents.

4. Copy the contents of the second file to the first.

5. Close the second file.

6. Open it again with a mode that deletes its content.

7. Copy the content of the auxiliary file to the first file.

8. Delete the auxiliary file from `/tmp/`.

Ex. 17: Write a function that receives a word as an argument and indicates in which position (in bytes) a word is in a file. Return just the position of the first ocurrence, if the word does not exists, return a negative number. Write a program that, with that function, receives a path and a word as arguments and prints the result of searching for the word in the file.

11.   How to write readable and clear programs

All programming languages have a different degree of **readability**. This is the easily a programmer would be to read the code written by another one, and understand what it is doing without a high level of effort. The language is not the only thing that affects the readability, the way one writes the code affects it at least as much, if not even more. Some advices to write more readable code are:

1. Use significative variable names, that is, instead of `a`, call variables things like `length`, `days`, or something that had anything to do with their meaning, with what they are.

2. Use function names that explain what the function is doing, in the same way, use names for the arguments that tell what they are, also.

3. Write the code indenting each new code block. More on this later.

Also, in C there are two tools of vital importance to increase the readability of code and that I am going to present to you now. One is the keyword `typedef`. This keyword allows us to **give an alias to a datatype**. That is, it allows to give other names to datatypes. One of the most practical uses to this is that it allows us to refer with a single word to refer to a struct, so we save the fact of writing `struct mystruct_s`, you can't avoid to tell me it's a relief after writing so many programs full of the word `struct`. Let's an example of this.

```c
#include <stdio.h>

struct point_s {
    double x;
    double y;

};

typedef struct point_s point_t;

int main(void)
{
    point_t p = {.x = 3.3, .y = 1.1};
    struct point_s q = { 1,3 };
}
```

Program 80: Definition of a type from a struct

This would be the most explicit way to do this. There is another alternative that we will see later on. From lines 1 to 4 there is nothing new, we simply declare the struct, it is line 6 that is the key, as you can see, we use the word `typedef` to define the type `struct point_s` is going to be called also `point_t`. Notice that, as you can see in line 14, we can keep using the old name of the types. All this being said, unless you want to symbolize something different to express a different meaning, it is better to use always the same name for the types. There is a more abbreviated way to do this, as a matter of fact, two, you can combine the sentence `typedef` with the struct creation in the same line. Doing this, you can choose to set a name for the struct or not, because its "real" name would be the one you define with `typedef`. Let's see it:

```
1  #include <stdio.h>
2
3  typedef struct point_s {
4      double x;
5      double y;
6
7  } point_t;
8
9  typedef struct {
10     point_t center;
11     double radius;
12 } circle_t;
13
14 int main(void)
15 {
16     struct point_s p = { 1.1, 2.3 };
17     point_t        q = { 1.2, 3.4 };
18     circle_t  origin = { .center = {.x = 0, .y = 0}, .radius = 1 };
19 }
```

Program 81: Different combinations of `struct` with `typedef`

The two declarations of these two structs are equivalent in the fact that they add a simple name to the structs, but with a nuance: the point struct keeps its struct name, so it could still be used to declare variables of this type as it is shown. The circle struct hasn't got an struct name, we have only used it to define a type. This is simpler but it presents a problem, when you do this and the compiler needs to tell you there's been an error, they would be different depending on the technique we use. With the struct `circle_t`, if we declare one function that receives an argument of this type and cell it with a different type, creating an error, the message would be:

```
main.c:14:18: note: expected 'circle_t' {aka 'struct <anonymous>'} but
    argument is of type 'int'
    14 | int foo(circle_t c){
```

As you can see, it sayd to you the name the type had, and then it tries to explian to you what it was originally, the problem is that, since we have defined the struct without a proper name, it has nothing to tell us, is an anonymous struct. And you may be wondering how can that be possible, well it is because in C you can declare structs without name like a variable to use them and throw them away. I am going to show it to you, but it is something I haven't seen in proffesional code so, like `goto`, keep in mind it exists, but it would be better if you do not use it.

```
1  #include <stdio.h>
2
3
4  int main(void)
5  {
6      struct { double x; double y; } temporal_point = { .x = 1, .y = 2 };
7      printf("This is a temporal point which is in [%1.2f, %1.2f]\n",
8          temporal_point.x, temporal_point.y);
8  }
```

Program 82: Anonymous struct

This is why we are able to define structs that have no name and and use this anonymous struct to define a type. My advice is that you always put a name to the struct to avoid the compiler to throw confusing errors. Compare this error of the compiler with the other error for the struct point that has a struct name regardless of `typedef`.

```
main.c:13:18: note: expected 'point_t' {aka 'struct point_s'} but
    argument is of type 'int'
   13 | void foo(point_t p){
      |              ~~~~~~~^
```

Finishing this with this topic: if you look closely, everytime I have put name to a struct I have ended it with _s, and each time I have written a `typedef`, I have ended it with _t. This is a convention, that is, it is something programmer do because of tradition, but it is not mandatory, not the compiler would throw an error or warning. Es suffix _s is less important, but I strongly encourage you to end all the types you define with `typedef` with _t, firstly: because it is very common, almost everybody does it and, secondly: because editors understand that any identificator (name, nasically), ends with a _t is a type, and they use it to give hints about what each thing is, for example, editor will colorize that word in the color that editor colorizes types automatically if it ends with _t.

Aside from this very practical use for us, there is another, it allows us to rename basic types, for example, we can rename the unsigned `char` as `byte_t`. In that way if we use our program to manage lists of bytes we can use this type and reader of our code will know what we're talking about.

```
1   #include <stdio.h>
2
3   typedef unsigned char byte_t;
4
5   void print_byte(byte_t b) {
6       byte_t current_byte = 128;
7       for (int ii = 0; ii < 8; ++ii) {
8           printf("%d", (b & current_byte) != 0);
9           current_byte /= 2;
10      }
11      printf("\n");
12  }
13
14  int main(void)
15  {
16      print_byte(110);
17  }
```

Program 83: Redefining basic types

Perhaps you do not understand all the code, the line 8, concretely, but simply look at how using a concrete type makes everything more readable. Also, it saves us to write unsigned several times. In the same way we did with the sign, we can define types with the const modifier, for example: `typedef const char letter_t`.

```c
1  #include <stdio.h>
2
3  typedef const char letter_t;
4
5  letter_t dictionary[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
       'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w
       ', 'x', 'y', 'z' };
6
7  int pos_of_letter(letter_t l){
8      for(unsigned int ii = 0; ii < sizeof(dictionary); ++ii){
9          if(l == dictionary[ii]){
10             return ii + 1;
11         }
12     }
13     return -1;
14 }
15
16 int main(void)
17 {
18     letter_t l = 'f';
19     printf("%c is letter number %d in the alphabet.\n", l,
           pos_of_letter(l));
20 }
```

Program 84: Example of definition of a type with const modifier

Even when this is posible, it is better to leave the const modifier as explicit, that is, define types for what u need, but write the modifier `const` when it is needed, instead of hiding it behind a typedef. It is a common practice to define unsigned types, but it is recommended that, unless the meaning of the type implies it can't be negative (for example a length), putting some prefix in the name of the type. For example, you may see, if you read some code, types like `uint32_t` which is a 32-bit integer (4 bytes) without sign.

And for final trick, with `typedef` this final trick is that, like arrays of different positions are in a way different types, you can define a type from an array. What is thif useful for? Sometimes a set of things of a certain size is a concept itself. Eight bits are a byte, seven days is a week and a hand has five fingers. Syntax for this is a bit different from what you may think at first instance.

```c
1  #include <stdio.h>
2
3  typedef unsigned char pixel_t[3];
4
5  int main(void)
6  {
7      pixel_t pixel = { 125,33,129 };
8      printf("This pixel has these values: Red = %hu, Green = %hu, Blue =
           %hu\n", pixel[0], pixel[1], pixel[2]);
9  }
```

Program 85: Definición de un tipo personalizado a partir de un array

As you can see, `[3]` is on the right of the name of the new type, instead of on the left, which would be the untuitive. Simply remember if you want to do this. Anyway, in the same way that with contance, a type that is an array is something that is better to leave explicit.

The other tool to add meaning to our programs, semantics, is the `enum`, or enumerated type. This is a type in C that allows us to assign correlative numbers to a set of names, the classic examples (so classic they are almost hackneyed) are the following: days of the week, months in the year, colors of the rainbow... The utility of this is that we can codify easily sets of names like those to names and use them to iterate a loop, index arrays, etc. Let's the example of the days of the week.

```c
#include <stdio.h>

enum week_days {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};

int main(void)
{
    printf("Todays it is: %d\n", SATURDAY);
}
```

Program 86: Basic example of enumerated type

If you compile and execute this you will see that it prints: "Today it is: 5". This is because when we write an `enum` like that, the first element gets the value zero, the next one, and so on... in this case from zero to six. And the enumerated types can be more useful if you couple them with arrays of types that help us to add meaning.

```
1   #include <stdio.h>
2
3   enum week_days {
4       MONDAY,
5       TUESDAY,
6       WEDNESDAY,
7       THURSDAY,
8       FRIDAY,
9       SATURDAY,
10      SUNDAY
11  };
12
13  const char* WEEK_DAYS_NAMES[] = { "Monday","Tuesday","Wednesday","
        Thursday","Friday","Saturday","Sunday" };
14
15  int is_today_weekend(int day){
16      return day == SUNDAY || day == SATURDAY;
17  }
18
19  int main(void)
20  {
21      for(int ii = MONDAY; ii <= SUNDAY; ++ii){
22          printf("Today it is: %s and\n", WEEK_DAYS_NAMES[ii]);
23          if(is_today_weekend(ii)){
24              printf("it's weekend!\n");
25          }else{
26              printf("it is not weekend.\n");
27          }
28      }
29  }
```

Program 87: Enum used alongside name array

Having an enumerated type allows us to write much more readable programs, which is what we wanted, because as you can see, the loop can read as: "For a `ii` from MONDAY to SUNDAY, if `ii` is in the weekend, print is it, otherwise, print it is not". Also, when printing the days, we can know the name of each simplt accessing the array of names. You may have noticed that both in the declaration of `ii` and in the body of the function I use the type `int`. This is because "inside" an enumerated type is an integer, but to write things more clearly, we can define an anumerated type in the same wat we defined an struct.

To see that the legibility of the code has improven, look at how the checking in the function may be read as: "if a day is SATURDAY, or SUNDAY, it's weekend". But the `enum` has more flexibility, we can define a value for each tag, or, this is more useful, define the value of the first, and the following ones will would take a correlative value. I am going to write a program with an example of both things, days of the week and months of the year. The next program shows all the alternatives.

```c
1  #include <stdio.h>
2
3  typedef enum week_days_e {
4      MONDAY = 2, TUESDAY = 4, WEDNESDAY = 6, THURSDAY = 8, FRIDAY = 10,
          SATURDAY = 12, SUNDAY = 14
5  } week_days_t;
6
7  typedef enum months_e {
8      JANUARY = 1, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
          SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
9  } months_t;
10
11 typedef enum seasons_e {
12     SPRING, SUMMER, FALL, WINTER
13 } seasons_t;
14
15 const char* season_names[] = { "Spring", "Summer", "Autumn", "Winter"
       };
16 const char* month_names[] = { "NOT USED", "January","February","March",
       "April","May","June","July","August","September","October","
       November","December" };
17 int is_today_weekend(week_days_t day) {
18     return day == SUNDAY || day == SATURDAY;
19 }
20
21 seasons_t season(months_t m) {
22     if (m >= MARCH && m <= MAY) {
23         return SPRING;
24     }
25     else if (m >= JUNE && m <= AUGUST) {
26         return SUMMER;
27     }
28     else if (m >= SEPTEMBER && m <= NOVEMBER) {
29         return FALL;
30     }
31     else {
32         return WINTER;
33     }
34 }
35
36 int main(void)
37 {
38     for (months_t ii = JANUARY; ii <= DECEMBER; ++ii) {
39         printf("It's %s and the season is: %s\n", month_names[ii],
               season_names[season(ii)]);
40     }
41 }
```

Program 88: Ejemplo final de enumerados

I have had to declare the `enum` in one line so it fitst in this pafe, but this has no effect whatsoever. Pay attention to how in the case of week days I have made each one to be an arbitrary value. Also, I have started months with one, I haven't defined the value of the rest, so they will take the next values: two, three... This definition has an implication, though, in the case of the days of the week I can't write the same loop to iterate over them **because now the values are not correlative**, but arbitrary. Also, if you look at the array of names for the months of the year, I have had to write in the start of the array a position that is not used, but it is needed so the indexes of the array names correspond to the enum values.

Opposing to `typedef`, the `enum` are somewhat situational, although, like the `do-while` or the `switch`, when u find that situation they're good for, they're very good for it.

11.1. Code style

Now we have talked about many tools of the language, it's time that we stablish, some rules to write our programs that I have left implicit and I'd like to write here. In this section we are going to see how the programs should be written beyond how they word, even, regardless of how efficient they are. When one works in programming, there are several moments in which you're going to be more time reading other programmer's code that writing new code. Consequently you must write your programs with the intent that they are readable for other programmers to read. Let's see an example of a code fragment that could be trivial or undecipherable depending on how it is written.

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3   int main(void)
4   {
5   int** a=malloc(10*sizeof*a);
6   for(int ii=0;ii<10;++ii){
7   (*(a+ii))=malloc(5*sizeof**a);
8   for(int jj=0;jj<5;++jj){
9   (*(a+ii))[jj]=10*ii+jj;
10  }
11  }
12  for(int ii=0;ii<10;++ii){
13  for(int jj=0;jj<5;++jj){
14  printf("%d ",(*(a+ii))[jj]);
15  }printf("\n");
16  }
17  for(int ii=0;ii<10;++ii){
18  free(*(a+ii));
19  }
20  free(a);
21  }
```
Program 89: Example of a program with a bad style

Try to read the program 89 and tell me if you know what it does. It is probable that, after five or ten minutes you can figure out it allocates a vector of vector, fills it up, prints it and frees it. As you can see, without spaces, without indenting them (indenting a line is putting space infront of it so they're displaced to the right), without whitelines between control structures and writing some lines next to the braces, it is almost impossible to read. Also, we haven't declared variables or constants that allow us to understand if the same value is like that by chance or symbolizes the same. I have also mixed arbitrarily different operators to access the vector. If we apply a series of improvements to how this is written, it will result in this:

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

typedef unsigned int uint_t;

int main(void)
{
    const uint_t ROWS    = 10;
    const uint_t COLUMNS = 5;
    int** matrix = malloc(ROWS * sizeof(*matrix));

    for (uint_t ii = 0; ii < ROWS; ++ii) {
        matrix[ii] = malloc(COLUMNS * sizeof(**matrix));
        for (uint_t jj = 0; jj < COLUMNS; ++jj) {
            matrix[ii][jj] = ROWS * ii + jj;
        }
    }

    for (uint_t ii = 0; ii < ROWS; ++ii) {
        for (uint_t jj = 0; jj < COLUMNS; ++jj) {
            printf("%d ", matrix[ii][jj]);
        }
        printf("\n");
    }

    for (uint_t ii = 0; ii < ROWS; ++ii) {
        free(matrix[ii]);
    }
    free(matrix);
}
```

Program 90: Ejemplo de programa escrito con un buen estilo

As you can see, we have introduced several improvements: operators are surrounded by spaces, `sizeof` is always used with parenthesis so they're more clear, we have used two constants to simbolize the size of the vector instead of repeating five or ten all the time, we hace used always the same way to access the vectors and, finally, we have written each instruction in a line with the correct indentation. Let's see which steps must be followed to write readable code.

11.1.1. Indentation

In the introduction to this section we have talked about indentation, and I have said to you that is it to precede some lines with spaces so they appear displaced to the right. This is one of the things you must do if you want to be taken seriously as a programmer, but there are several things about how to do it. The first decission you have to take is if you indent with tabulators or with spaces. A tabulator is a special caracter that indicates the text editor to align things to the next column in the display. The avantage of this is that tabulating with them you can make the editor you're using to take the columns to which the tabulators align as any size of spaces. This character is inserted with the tabulator key, which is the one that has this symbol: ⇤⇥ or has the legend "tab" in it. Another option is to indent with spaces, that is, instead of using that special character, you use a set number of spaces. In general, you do not hit space several times, but yo configure your editor in such way that when you hit the tabulator key, it inserts the spaces you have configured as "tab size" or "indent size".

This is one of things that would make programmer fight each other like religious zealots or hooligans of a football team, so the decission is yours. Nevertheless; my honest opinion is that spaces are better since they give you more liberty to align longer lines that must be broken. Also, ansures that the code is going to be shown correctly in all the editors, since there is not configuration about how a space must be shown, but people may have different tab size configured.

The general rule is that code blocks must have their instructions indented a level more than where they are define. You can see how I did this in the programs I have shown up until now. Each tabulation level must be the same as the others, that is, if you have chosen four spaces, you must indent always with four spaces, never mix. Also, if you're using tabulator characters, an indentation level shouldn't be more than a tabulator.

Indentation is also important when a line is very long. You may be wondering what sense does it make to limit the lenght of the lines of code is we're using editors that allow us to make them as long as we want. This is so for several reasons: when lines are too long they tend to be more difficult to read, they avoid you from having several files opened at the same time and, also, in the improbable, but possible case you need to print your code, it would be difficult to do it properly. The last case does not affect you, but had affected me personally in several of the programs I have written as examples. The length of line of code in C is generally around 80 characters, ideally always less than that. To allow you to have in image, the lines I have inserted in the manual have around 71.

And what to do when the line is simply too long? There arw ways to write the lines in different styles. This way changes depending on how in a program you have the problem, so let's see some examples on how to fix it in several places.

The first case is when you use functions with many arguments, or when the names of those arguments are too long. A classic example of this are the calls to `printf`, but you can see it with other functions. For example, imagine a program that prints a somewhat long message for the user, it could be like this:

```
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("This is an important message, stay hydrated, drink water
              and avoid extreme heat situations.\n");
5  }
```

Program 91: Long printing instruction

As you can see, the line does not fit in the screen, what tends to be done in these cases is to split the constant in several. As long as you write them together or separated only by whitespaces (both spaces, tabs or newlines), C will see them as just one. Let's see the result:

```
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("Este es un mensase importante, "
5              "por favor, mantente hidratado,"
6              " bebe agua");
7  }
```

Program 92: Splitting a long printing instruction

Notice that, although we are splitting it, we must include the spaces between words. Also, then you split strings in this way you tend to write them all in the same level of indentation.

The next example is a function with several arguments, or with very long ones, let's image a program like this one:

```
1  int main(int argc, char** argv)
2  {
3      int list[] = {};
4      int the_position = 10;
5      int* res = insert_at(list, ARRAY_SIZE(list), 0, rand());
6      print_array(res, ARRAY_SIZE(list) + 1);
7      free(res);
8  }
```

Program 93: Example of a call with several arguments

Since I am not interested in what the program does, but only in the format, I am not going to include the function definition, but the function `insert_at` creates a new array with the element given as last argument in the position that the second argument indicates. If we wanted that the line where that call occurs were separated in several ones, we could do do in these styles:

```
1  //1
2  int* res = insert_at(list, ARRAY_SIZE(list),
3                                      0, rand());
4
5  //2
6  int* res = insert_at(list,
7                       ARRAY_SIZE(list),
8                       0,
9                       rand());
10
11 //3
12 int* res =
13     insert_at(list, ARRAY_SIZE(list), 0, rand());
```

Program 94: Splitting of call with several arguments

In the first example we make a cut in only one argument and we write the rest in a new line one indentation level to the left of where it would be if we wrote it in the same line. This method is useful when we surpass the character limit by just a few lines. The next is my preferred one and basically we write each argument in a new line and on the same level. In the last one, we take advantage of the fact that we are assigning the result of the call, we use this operation to perform the split, with a new line and an indentation level to the left. The three could be combined, but I like the second because it disallows weird alignments and also saves the more space. Some people may think it's overdoing things.

The next place where you may need to split a line is the declaration of a function with several argument, for example, a function that prints a date given the year, the date and the month and an argument that tells us if we want DD-MM-YYYY format or YYYY-MM-DD.

```
1  void print_date(unsigned short day,
2                  unsigned char  month,
3                  unsigned int   year,
4                  int            order);
```

Program 95: Splitting of function declaration

In these cases you tend to write all the arguments in a different line, also, I have taken the effort in alignining the names and the type of the arguments to create a species of table, which facilitates the reading. This is merely stylistic and optional.

In other places where you can encounter these situations is in lists of several types, I am applying here the word list in a loosely way, I am referring to the successions of things separated by commas that are between braces: initialization lists specially, in general you can apply the same techniques that with the first method when we split the call of a function.

11.1.2. Spacing symbols

In general terms most of the whitespaces what we write in our programs are to make it easier to read, because C is a language designed so the whitespaces don't matter as much. Normally, mathematical operators, logic, or of any kind should be surrounded by spaces, that is: it is preferred `a = b + 10;` to `a=b+10;`, it surrounds the assignment operator and the plus operator with spaces. Also, in declarations the asterisk must be always next to the data type, or to the name of the variables. It is a question of style, but I recommend to stick them to the name of the variable or argument. What is important is to do it in the same way everytime inside the same project. For example:

1. Space operators: `int var = a * 3 + ii`

2. Asterisks should be next to the name of the variable or argument in declarations or to the one they're dereferencing:

   (a) `double *var1;`

   (b) `int function(int *arg1, void **arg2);`

   (c) `int a = *ptr1 + *ptr2;`

Also, after all the commas in initialization lists, the semicolons in the `for` loop or the arguments of a function there must be an space and spaces must also surround the parenthesis of control structures, (note, function calls are not a control structure), that is:

1. `for (int i = 0; i < 10; ++i) {`

2. `int list[] = {1, 2, 3, 4, 5, 6};`

3. `int res = pow(var1, var2);`

11.1.3. Brace style

There are several combinations of style between the braces that define code blocks that make the body of functions, loops conditionals and other control structures. There are two fundamental styles: braces K&R and Allman. The first ones are in the same line that the structure whose body they are opening, and the second ones in the next line.

```
1  int main(void) {
2      return 0;
3  }
```

Program 96: Example of braces in K&R style

Braces in Allman style look like this:

```
1  int main(void)
2  {
3      return 0;
4  }
```

Program 97: Example of braces in Allman style

There are other styles that combine the indentation and the braces in a different way, but they're not relevant at this level. A way that is common is to use Allman style in the function declaration (that is a code block of the first level) and K&R in the rest of the blocks. The most important thing is to use one style or the other, or a combination of them and maintain them in all the project. In case of adding code to an existing codebase, apply the old rule of "when in Rome, do as Romans do", that is, follow the style that is already in place in existing code of the project.

11.1.4. Declarationf of struct, variables or enums

Up until now, each variable declaration has been written in a single line, and each member of the structure. This is not needed, you can declare all the variables of the same type in the same line, even initialize them. Let's see a simple example. We are going to rewrite the program 58: Reserva, uso y liberación de un vector de vectores but I am going to skip part of it that hasn't changed.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int rows = 10, columns = 5, **matrix = NULL;
7      matrix = malloc(rows * sizeof(*matrix));
8
9      for (int ii = 0; ii < rows; ++ii) {
10     //resto del programa...
```

Program 98: Declaration of variable in the same line

As you can see in line 6, we can declare variables in the same type in the same line, separating their names with commas. Also, we can even initialize them. If you're going to declare pointers in the same line of other variable that are not, the asterisk that indicates the variable is a pointer goes next to the name of the variable, as you can see with `matrix`. In the case of initializing the pointer, it could be in the same line, but I have moved it to another because otherwise it would have been too long.

With the structs you can do the same, that is, declare all the member of the same type in the same line, with the enumerated types it is the same, although it is convention that each member of a struct is in its own line.

```
1  struct point_s {
2      double x, y;
3  }
4
5  enum week_days {
6      MONDAY,
7      TUESDAY,
8      WEDNESDAY,
9      THURSDAY,
10     FRIDAY,
11     SATURDAY,
12     SUNDAY
13 }
```

Program 99: Declaration of member of a *struct* in a single line

11.1.5. Name convention

One of the most important things you must avoid is the use of so called **magic numbers**. Those are the literal values that are included in the code and are not explained, they are assigned to a variable and, in the worst case, they're used in several parts of the code. If you rescue the example of bad code of the program 89, you will see that I never define variables to define the rows or columns of the matrix. I did this intentionally, because it is a bad thing to do. As you can see in the example of good code, the first thing I do is to define variables for these concepts. In general, we must define variables for all the concepts or values that in your code **specially if it appears several times**.

Also, there is a series of conventions that we are going to gather here about names. When I explained the name that a variable can have, I told you that they could be written in camel case or snake case. In C it is usual to use the second, that is: `my_var`, so it will be the one I'll use in the examples, and that I have used up until now. On the other hand, constant should be written in uppercase, that is: `const int LENGTH = 10;` is better than `const int length = 10;`.

In general, the types end in _t, as we commented, either if they're renaming of basic types or creation of structs. It is normal to omit the name of the enums, that is, it is usual to write:

```
1  typedef enum {
2      MONDAY,
3      TUESDAY,
4      WEDNESDAY,
5      THURSDAY,
6      FRIDAY,
7      SATURDAY,
8      SUNDAY
9  }week_days;
```

Program 100: Example of a enum with typedef

Function must have descriptive names, for example, it is better to write:

```
1  int multiply_array_scalar(int *array, int array_size, int scalar);
```

Program 101: Example of descriptive function

than

```
1  int multi(int *a, int s, int n);
```

Program 102: Example of non-descriptive function

In the first case it is clear what the function does and that each argument is, at first sight. In the second you'd have to go to the implementation to see what it does.

12. What is behind compilation

Up until now we hace said that creating a binary from a source code file is compiling. Although this is correct, is it not totally true. There are several processes involved in what we call compilation:

1. Preprocessing: Prepares the code to be compiled, this includes deleting comments, expand macros, execute directives and delete whitespaces. More about macros, directives and comments later on.

2. compiling: Creates the binary of each source code file that is used in the project.

3. Linking: It generates the executable binaries properly said, to do so it uses the binaries generated in the last step and creates links between them, regardless it being joining them in the same file or simply linking one to the other indicating were it is.

This intorduces a series of new concepts I will explain now, but the fundamental here is that this is the section in which you're going to learn how to make programs with more than a source code file. This is useful because, as you saw, just when a program a bit complex, going up and down over the file starts to be confusing, specially to modify things. Because of this, every serious project in the C language contains several source code files. Some projects may reach the tens of thousands of files. Let's see the three steps described before.

12.1. Preprocessing

Preprocessing is the most discrete step of the compilation process, because every source code file that is compiled passes through it and we do not have to do anything. In the introduction of this section I have said that several tasks are performes, first of those being the deleting of comments. It is time I explain to you what those are. A comment is a tool that allow us to introduce any arbitraty text in our code so we can annotate things, explain the code or things like that. Let's see an example. I am going to include comments in the program 37: Calculating the distance between two points using structures.

```c
1  #include <stdio.h> //for printf
2  #include <math.h>  //for sqrt
3
4  //We define a bidimensional point
5  struct point_s {
6      double x;
7      double y;
8  };
9
10 int main(void)
11 {
12     struct point_s A;
13     struct point_s B;
14
15     A.x = 1.1;
16     A.y = 3.2;
17     B.x = 2.3;
18     B.y = 5.4;
19
20     double diff_x = A.x - B.x;
21     double diff_y = A.y - B.y;
22     /*Remember that the distance
23     between two points is the square
24     root of the addition of the substraction
25     of their coordinates squared*/
26     double distance = sqrt(diff_x * diff_x + diff_y * diff_y);
27
28     printf("P1 : [%f, %f]\n", A.x, A.y);
29     printf("P1 : [%f, %f]\n", B.x, B.y);
30     printf("Distance: %f\n", distance);
31 }
```

Program 103: Example of program with comments

As you may have guessed, the green texts are the comments. It is clear that they can be any text you may want, and that they have nothing to do with C code. The preprocessor will delete them before passing the code to the compiler. There are two kinds of comments:

1. One line comments: they start with //, they're comments that make the processor to delete everything after the slashes, these included, up until the end of the line. You can put several in correlative lines.

2. Multiline comments: they start with /* and **end with** */. The preprocessor ignores the new lines and any character that is written up until the end of the comment, as you can see in the example.

In general, you can put the comments you deem necessary, specially, it is recommended to comment functions that are very difficult or thins that are not evident.

The next step is the expasion of macros, and you may be wondering that a macro is. A macro is a symbolic contant that we define in the code and preprocessor will substitute it by its value where we write it, let's see an example.

```
1  #include <stdio.h>
2
3  #define LIST_LENGTH 100
4
5  int main(void)
6  {
7      int list[LIST_LENGTH];
8      for(int ii = 0; ii < LIST_LENGTH; ++ii){
9          list[ii] = ii;
10     }
11 }
```

Program 104: Macro creation

In line 3 you can see the only nnew thing we have here, the directive `define`. Directive start with a ound sign (`#`). And we're about to unveil one of the misteries in our programs, the lines that start with `#include` are directives, but let's avoid getting ahead of ourselves. This directives defines (I know, shocker) a symbol that the preprocessor will substitute with the one we have set in the definition of the macro. In this case, the symbol `100`. A common use of this is, like in the example, defining the size of arrays. Does this mean that every array must have its corresponding macro? No, only if it's going to be used in several places or has a meaning, for example the size of a list of tests.

Since macros are substituted **before** compilation, you can, indeed, define the size of the array and initialize it at the same time. This has and advantage: when you initialize an array, but you do not use all the elements, the remaining are initialized to zero automatically, so you can write your programs like this:

```
1  #include <stdio.h>
2
3  #define LIST_LENGTH 65536
4
5  int main(void)
6  {
7      int list[LIST_LENGTH] = {};
8      for(int ii = 0; ii < LIST_LENGTH; ++ii){
9          if(list[ii] != 0){
10             printf("There are elements that aren't  0.\n");
11             break;
12         }
13     }
14 }
```

Program 105: Macro use with arrays

I explained to you that you cannot use a variable to define the size of an array that you want to initialize with an initialization list. This is the case because the compiler, which would need to know if the initialization list is too big or too small, cannot do so because the values of a variables is unknown until we execute the program. You may be wondering if a variable with the `const` modifier would be an alternative. No, it will not compile with a variable also, because while it is marked as a read only variable, there are some mechanisms which can modify the value, some totally normal and others as consequences of abuses of the language or bad practices. Regarding the fact that when the initialization list is smaller than the size of the array the remaining positions are initialized to zero, you can make a test. Compile that last program an execute it a couple times, you'd see it never prints the message, now, delete the initialization of the array, leaving line 7 as `int list[LIST_LENGTH];`, compile and execute again, and you'll see most often than not it prints the message.

Macros have an incredible power, because they can receive arguments. You may have noticed that there are things we do very often, like for example square a number, we use the function `pow` for that, but it needs to compile with the math library in the same way that `sqrt`, or writing `var*var`, with a macro, we can do that more legible and without using functions.

```c
1  #include <stdio.h>
2
3  #define SQUARE(a) a*a
4
5  int main(void)
6  {
7      for (int ii = 0; ii < 10; ++ii) {
8          printf("%d^2 = %d\n", ii, SQUARE(ii));
9      }
10 }
```

Program 106: Macro with arguments

As you can see, this program works as intended. Being this so practical, why aren't macros used for everything? The reason is that they can become a very difficult thing to understand or fix if there is an error. They're just a symbolic substitution, if you created a set of macros that repeated a block of code a set of times, in the fashion a loop would do, in case you got an error you'd have to look for it in lines that you cannot see because they haven't been substituted yet, and you'd have to compile each time to catch the marcro whose substitutiones is creating the error. Also, there may be problems because macros do not understand about types or priotity of operations, let's see an example of the risks.

```c
1  #include <stdio.h>
2
3  #define SQUARE(a) a*a
4
5  int main(void)
6  {
7      for(int ii = 0; ii < 10; ++ii){
8          printf("%d^2 = %d\n", ii, SQUARE(ii+1));
9      }
10 }
```

Program 107: Example of error because of a macro

If you read this program, you may expect that it prints: 1, 2, 4, 9... but it is not what it happens, it prints: 1, 3, 4, 7, 9... and this is like that because this macro is not properly written, make the mental exercise to substitute `a` in the macro by `ii+1`, you'd get that the result is `ii+1*ii+1`. This is not what we intended. The solution for this is writing the argument of the macro between parenthesis, that is: `#define SQUARE(a) (a)*(a)`, but even when it is an easy fix in this case, let this example server as a warning on how dangerous macros can be when used recklessly.

Also, there is a version of the macros that allows to write a text string the argument that is put inside it. Let's see how it is done and used.

```c
#include <stdio.h>

#define PRINT_INT(a) printf(#a"=%d\n", a);

int main(void)
{
    for(int ii = 0; ii < 10; ++ii){
        PRINT_INT(ii);
    }
}
```

Program 108: Use of macros with strings

As you can see, when you write `#a` inside the definition of the macro, we're asking it to substitute `a` by `"a"`. As we saw in the program 92 when we write several string literals separated only by whitespaces, they're effectively treated as a single one. Later we have written `a` so it is substituted normally.

This is an application of macros that transform the arguments in strings, in general, when you use that feature in a program intensively you may want to create a macro that stringifies (converts to string) the arguments, and, most importantly, allows to write the value of macros that are not strings (numbers) as strings.

```c
#include <stdio.h>

#define STR_HELPER(a) #a
#define STR(a) STR_HELPER(a)
#define PRINT_INT(integer) printf(STR(integer)" = %d\n", integer)

int main(void)
{
    for (int ii = 0; ii < 10; ++ii) {
        PRINT_INT(ii);
    }
}
```

Program 109: Macro to stringify

Here it is shown how the macros are used to print firstly the name of the variable and then then the value. This is, also, useful when you have macros with numeric values you want to turn into a string, a case would be this:

```
1   #include <stdio.h>
2
3   #define STR_HELPER(a) #a
4   #define STR(a) STR_HELPER(a)
5
6   #define ARRAY_SIZE 10
7
8   int main(void)
9   {
10      int array[ARRAY_SIZE];
11      printf("The size of the array is "STR(ARRAY_SIZE)".\n");
12  }
```

Program 110: Converting numeric macros to string

The detail I want you to pay attention to is the fact that, in this case, since it is a macro and not a variable what is passed to STR, it is substituted by its **value** and not its name. You can use this to define strings, regardless of its use alongside `printf` or not.

The next directive we're going to look into is, at last, `include`. This is the directive we have used to be able to use a variety of functions, like `printf` or `malloc`. This is what the directive does: it embeds the content of one file into other. Yes, you have read it well, when you write #include <stdio.h>, the only thing you're doing is to paste in this file the contents of another file, in this case `stdio.h`. It is weird it is a file with `.h` extension, if all the other programs we have written are in a file ended in `.c`. This is because this files is a **header** file, in this kinf of files, of which we will see more later on, only definitions are written, that is: function declarations, global variables, new types...

Nevertheless, at the moment, you can experiment with this directive writting these two files: `main.c`, which we already had, and `other.c`. In the first one, write this:

```
1   //main.c
2   #include <stdio.h>
3   #include "other.c"
4
5   int main(void)
6   {
7       int a = 10;
8       printf("a is now: %d\n", a);
9       multiply(&a, 2);
10      printf("a is now: %d\n", a);
11  }
```

Program 111: Example of `include` directive, main file

As you can see, when including `other.c`, we're using quotes instead of the less than and greater than sign we used with the other include directives. This is because when the directive receives the name of a file surrounded by less than and greater than, the C preprocessor looks for the file in certain directories the system has configured where header files go. When you use quotes, it looks for them in the directory where the source file is. Because of that, when you use files that are not in those special directories, you must take this into account. The file `other.c` must have this content.

```
1  //other.c
2  void multiply(int* a, int b){
3      *a *= b;
4  }
```

Program 112: Example of `include` directive, included file

If you look at the content of both files, we're simply "putting out" the function to other file and we have used the include directive from the main file so all the content of the secondary file gets written in the main one. Nevertheless; a golden rule is that you should never include source code files, only header files. But, to do so, we must discuss fristly the next two steps of compilation.

Before arriving there, nonetheless; I want you to see a couple of very interesting directives, `ifndef`, `ifdef` and `endif`, that go together. Their names is more or less self explanatory, but they do this: these directives check if a macro is defined and, according to it being defined or not, the code between `ifdef`/`ifndef` and `endif` will be included or not. Let's see an example.

```
1   //main.c
2   #include <stdio.h>
3
4   int main(void)
5   {
6   #ifndef MY_MACRO
7       printf("Hello, world!\n");
8   #else
9       printf("Good bye, world!\n");
10  #endif
11  }
```

Program 113: Uso de directivas `ifdef` e `ifndef`

Here you can see that we use `ifndef`, verb!else! and `endif`, I think itis easy to see, but simply, when `MY_MACRO` is **not** defined, the code resulting from the preprocessing will print "Hello, World!"; when is is defined, it will print "Good bye, world!". Macros can be defined from the command line when compiling (mind that we are not interested in their value, just if they're defined or not) so this allows us to shape our compilation to different environments. To define a macro in the compilation command with GCC simply add `-DMACRO_NAME=value`, for example, in the case I have just shown: `-DMY_MACRO=0`. Test it, compile the program with this command: `gcc -o main.exe main.c` and execute it, it will print "Hello, world!", if you compile it with `gcc -o main.exe main.c -DMY_MACRO=0` it will print "Good bye, world!". If, instead of `ifndef`, we used `ifdef` the behaviour would be the opposite.

For example, it is used to make some printing orders to exist or not depending on how we compile the program, to make some messages to appear when compiling the program to test it, but making them not to appear in the program version that would get sold, for example. These directives are important for something we will see in the next section.

12.2. Object compilation

The next step is the compilation, *stricto sensu*. It is a very simple step, but with very interesting implications. What you have been done up until now is compiling a program with just a source file. But if you executed the compilation command in another way:

```
$ gcc -c <source code file>
```

This will generate a file with the same name of the source file, but with the `.o` extension. This is **Compiled Object File**, or object file for shorts. These files are an intermediate point between the source code in C and the executable, it is so because these files still have information about symbols (variables, functions...). If you compile with thisoption the program Hello World we wrote the first time, you'd see you **cannot** execute this object file, but those files are the ingredients we will use in our crucible to build an executable with several files.

To do so I want you to remember the files `main.c` and `other.c` and **delete** the lines where you included `other.c`. If now you try to compile to file `main.c` with the order `gcc -c main.c` the compiler will throw a warning like this one:

```
main.c: In function 'main':
main.c:8:5: warning: implicit declaration of function 'multiply' [-
    Wimplicit-function-declaration]
    8 |     multiply(&a, 2);
      |     ^~~~~~~~
```

It tells to us that we have declared implicitly the function `multiply`, that means that the compile hasn't found the definition of the function in the source code, that is, it warns you that this functions is pending on existing as it has been used (name, return type and arguments). Maybe this looks like madness, but it is true, in C you can declare functions simply by calling them because it is expected that they **they are in other object files**. That's the reason that object files still save information about names of functions, because in that way, when you stick them all together, you get the whole puzzle, the executable.

Now, we must create the object code of the other file, simply executing the same command, but with `other.c`. Now you will have two files: `main.o` and `other.o`. We already have all the pieces, but to assemble them together, you would simply execute:

```
$ gcc -o main.exe main.o other.o
```

This will generate an executable that you can execute normally and you will see that, effectively, it works. As you may have guessed, it is a very bad idea that when you compile a source code file in which there are several functions the compiler cannot tell you if you're using them correctly and they exist. If you have experimented with the exercises, you would have seen it is very easy to commit errors and use functions with the wrong arguments or misspell the name of a function. The compiler is your best ally to find them. Is in solving this where the header files and the source code files come in place. As we saw in section 7, you can declare functions in a place and define them in another, and this is where that is very useful. All function can be declared in a header file, in such a way that the compile **knows** the signature of the functions and can generate the code checking againt those definitions. To do so, we're going to write the header file that goes with `other.c`, that is, `other.h`. It's very easy, as we only have one function, the file would be like this:

```
1  //other.h
2  void multiply(int* a, int b);
```

Program 114: Header file

You need to change `other.c`, simply **including** the header. We do things like this because, if we make a mistake defining the function, for example, imagine that we forget the asterisk, the compiler will tell us that we have redefined the function because we have declared it with a set of arguments but defined it with another, that would be like two functions with the same name, which is not permitted.

```
1  //other.c
2  #include "other.h"
3  void multiply(int* a, int b) {
4      *a *= b;
5  }
```

Program 115: Definition file with included header

Finally, in `main.c`, we will include the header file too:

```
1   //main.c
2   #include <stdio.h>
3   #include "other.h"
4
5   int main(void)
6   {
7       int a = 10;
8       printf("a is now: %d\n", a);
9       multiply(&a, 2);
10      printf("a is now: %d\n", a);
11  }
```

Program 116: Main file with included headers

Now, to generate the binary, we simply have to generate both objects and later on the executable with these orders:

```
gcc -c main.c
gcc -c other.c
gcc -o main.exe main.o other.o
```

If you pay attention, you'd see the the compiler no longer throws the warning about the use of the function without defining it. But there is a problem, the code `other.h` would be embeded in all the files that use the function, because in all of them the directive would be present. If we leave this like it is, we couldn't use the function in another files, because the compiler would see this as a redefinition. I will include here all the files of an example of this.

```
1  //point.h
2  struct point_s{
3      double x;
4      double y;
5  };
6
7  typedef struct point_s point_t;
8
9  double distance(const point_t* a, const point_t* b);
```

Program 117: Redefinition example – `point.h`

```
1   //point.c
2   #include "circle.h"
3   #include <math.h>
4   #include <stddef.h>
5
6   double distance(const struct point_s* a, const struct point_s* b) {
7       double res = 0.0;
8       struct point_s origin = { .x = 0.0 , .y = 0.0 };
9       if (NULL == a) {
10          a = &origin;
11      }
12      if (NULL == b) {
13          b = &origin;
14      }
15      double diff_x = a->x - b->x;
16      double diff_y = a->y - b->y;
17      res = sqrt(diff_x * diff_x + diff_y * diff_y);
18      return res;
19  }
```

Program 118: Redefinition example – `point.c`

```
1   //circle.h
2   #include "point.h"
3
4   #define PI ((double)3.141592)
5
6   struct circle_s {
7       point_t center;
8       double  radius;
9   };
10
11  typedef struct circle_s circle_t;
12
13  double area(const circle_t* c);
14
15  double diameter(const circle_t* c);
```

Program 119: Redefinition example – `circle.h`

```
1  //circle.c
2  #include "circle.h"
3  double area(const circle_t* c) {
4      return c->radius * c->radius * PI;
5  }
6
7  double diameter(const circle_t* c) {
8      return 2 * PI * c->radius;
9  }
```
Program 120: Redefinition example – `circle.c`

```
1  //main.c
2  #include <stdio.h>
3  #include "point.h"  //for using points
4  #include "circle.h" //for using circles
5
6
7  int main(void)
8  {
9      point_t a = {1.1, 2.3};
10     point_t b = {4.1, 3.3};
11     printf("Distance between a and b is: %f\n", distance(&a, &b));
12
13     circle_t circle = {a, 1};
14     printf("The circle has an area of: %f\n", area(&circle));
15 }
```
Program 121: Redefinition example – `main.c`

If you try to build the object files with these commands (remember, header files are not compiled):

```
gcc -c point.c
gcc -c circle.c
gcc -o main.exe main.c circle.o point.o -lm
```

You can see that in the last command we have created the executable indicating the name of the source code file for `main.c` instead of creating firtly and object, this is a way to save the step of creating the object for the main file (the one that contains the `main` function) you will see that the last step throws error about redefinitions. This is because, if you follow the "trail" of includes, you'd see that `main.c` includes `circle.h` and that it includes `point.h`. On the other hand, the very `main.c` includes `point.h`, this makes that the content of the later to be duplicated. In this example it could be very well solved simply deleting the inclusion of the point header in the main file, but we can't do that, because deleting the inclusion would make impossible to use the structures defined there in the main file unless we include again `point.h`, making mandatory to keep tracking any change in the includes for consequences. Also, it is adviceable that each file includes all it needs even when the headers are included in other ones already in use so it is clear what it is related with. When a project is very big, tracking this kind of duplicate includes is almost impossible, this is when the so called **include guards** come in.

These are simply the use of directive of the kind `ifndef endif` to make that, if a header file is included more than once, the repetitions of the file would be empty files. It is customary that header files in the project have one, so when you use several files for your programs I advice your to start using them now. Let's see how to add an include guard to `point.h`.

```
1  //point.h
2  #ifndef POINT_H
3  #define POINT_H
4
5  struct point_s{
6      double x;
7      double y;
8  };
9
10 typedef struct point_s point_t;
11
12 double distance(const point_t* a, const point_t * b);
13
14 #endif
```

Program 122: Example of include guard

As you can see, what is done is to enclose all the content of the file in a preprocessor conditional. If the macro `POINT_H` if not defined, we will define it and, with it, all the code of the header. In this way, when the file is included the second time, since said macro is already defined, the `ifndef` wouldnt be true and, to the compiler, this file will be empty. The name of the macro tends to be the name of the file, substituting the dots by underscores, if you used directories inside your project, ideally the macro should have the complete path from the root of the project, for example, if the file is in `project/lib/math/geometry/include`, the name of the macro for the include guard should be: `LIB_MATH_GEOMETRY_INCLUDE_POINT_H`.

12.3. Linking

Linking is the final step to create an executable. In it, what it is done is to assemble all the object files and different **libraries** needed for the creation of the executable. You already know what object files are, and you have already used several of them to create an executable, now we're going to take a deep look into libraries.

A library is, from a conceptual point of view, a set of functionalities that are compiled and distributed as a whole in a packet that the programmer that wants to use it can incorportate to his programs. The main advantage is that source code of a library is practically impossible to recover from the library itself, and, specially, in a comprehensible manner. This is an advantage because the source code is subject of intellectual property and, if it is common to have open source projects, many commercial libraries are distirbuted without access to the code.

Also, libraries move the responsibility of the code inside them to the vendor, relieving the user of them of the workload of creating the binary once the code there fails. They're an esential tool to distribute software. As a matter of fact, you have already used libraries, all the functions you have used up until now that came simply from using the include directive (`malloc`, `printf`...) are located in different libraries that **are distributed as part of your operating system**. When the operating system updates, these libraries may change and therefore their functionality may change also. This may implicate you need to recompile your programs to see those changes or not, depending on which type of libraries you use, there are two kinds:

1. Dynamic libraries: Those are the ones that are loaded into memory when the program executes, hence the name, since they are only loaded when they're needed.

2. Static libraries: Those are the ones that combine with the binary in compilation (linking) time. This implicates that when a library of this kind changes, the binaries that use it need to be recompiled to update themselves.

The libraries, like the executable, **are managed by the operating system**, that is, the executables that use libraries would ask the operating system (without the programmer neded to do anything in the source code) to load them when it is needed. Also, it is the one that manages the versions, allowing the programmers of a binary to indicate that their executable would only work with certain versions of the libraries it uses.

In general, when libraries are involved, either created or used, the programmers use certain toold that automate the creation of the object files and the libraries themselves. Nevertheless, I am going to show you how it would be done, "by hand".

Let's suppose we want to create a library with our point struct and our circle struct and that we're going to call it `libGeometry`. It is convention that all libraries start by `lib`. The first step is to generate the object code, because we use it both for creating executable and libraries.

```
$ gcc -c point.c
$ gcc -c circle.c
```

Once you have done that, we create the library, to do so we will use this command:

```
$ gcc -shared -o libGeometry.so point.o circle.o
```

We use the option `shared` to indicate that we're building a dynamic library. This will create our library file: `libGeometry.so`, now we can build the executable using the library instead of the object code directly. You can do so with this command:

```
$ gcc -L. -Wl,-rpath=. -o main.exe main.o -lGeometry -lm
```

This command is a bit complicated, the option `-L` allows us to indicate in what directory are the libraries for the compilation, this is done indicating the dot directory `.`, that means this very directory. On the other hand, the option `-Wl,-rpath=` allows us to indicate where the library file must be looked for in **execution time**, in the same fashion, we write a dot. The rest of the command is the same, but we add the linking of the libraries: our own and the math library, with option `-lm`. Now, if you execute the program with the command `./main.exe`, it will work correctly. You can check that the library links dynamically, to do so, delete the file, you can do it with the command `rm libGeometry.so`. If you try to execute the program, it will not work, because it will try to load the library when executing the program.

In the case of a static library, these are created in a simpler way, compile the the object like before and create the static library:

```
$ ar rvs libGeometry.a point.o circle.o
```

Now we can compile including the libraty (we need to keep adding the `-lm` because we use the function `sqrt` that is in the mathematical library):

```
$ gcc -o main.exe main.o libGeometry.a -lm
```

To check it is static, delete `libGeometry.a` and you will see that the executable keeps working. This is because the library is embeded in the executable when you create it. This makes the result similar to use all the object files, but it still allows the easy **distribution** of software. Also, this allows us to create conceptual software units that are bigger in object files and that are conviniently in a single file, simplifying our processes of creating and distributing programs.

13.   Standard library functions

The standard library of C is a library that is included in all the programs that are compiled in C in Linux. This is like that because it contains most of the functions that are needed to perform basic tasks, for example: `malloc` and `free` are in it. Although there is a header called `stdlib.h`, most of functions that can be used without linking additional libraries are in the standard library. We're going to talk about some of these functions and show why they're needed even in the most basic programs.

13.1.   Memory management

Although we have seen the most basic functions for memory management: `malloc` and `free`, there are other functions that are useful and it's good to know them. These are `calloc`, `realloc` and `memset`. The first two enable us to allocate memory and the last one is useful to set the same value of all the bytes in a memory zone. They're used often in programs with several memory operations.

The first of them: `calloc` is a function that allows us to indicate the memory allocation os several blocks of a concrete size, that will be allocated in a contiguous zone. To start, let's see the signature of the function:

```
1  void *calloc( size_t nmemb, size_t size );
```

Program 123: Signature of function `calloc`

As you can see, in the same way that its "sibling" `malloc`, it returns a pointer to `void`, that can be assigned to any type of pointer. Nevertheless, it receives two arguments: the numbers of elements you are going to allocate space for and the size of the elements. If you're thinking that a call to this function is equivalent to calling `malloc` multiplying both arguments, you'd be right, be there is a **caveat**: the memory reserved with `calloc` will be initialized with **zeroes**. This makes some programmers to use a call to `calloc` with the first argument being one to allocate a memory zone that is initialized to zero.

The next function, `realloc`, is more interesting, it is a function that allow us to **resize** and, automatically move, if needed, a memory zone, the signature is as it follows:

```
1  void *realloc(void *ptr, size_t size);
```

Program 124: Signature of function `realloc`

As you can see, it receives a pointer as first argument, this is a pointer to the memory zone we want to resize and, as second arguments, the new size, **in bytes** of the memory zone. In this case there are two possibilities, you are either enlarging the zone or shrinking it. The function gives no guarantee in any case that the memory region is going to stay in the same place, so you must check it hasn't returned `NULL` and save the value again because it may have changed. Also, if the pointer that the function receives is a null pointer, it simply behaves as `malloc`, this is useful then you use them in loops without needed to use `malloc` before starting the loop. To show the use of this function we will write with it the program 56 where we showed the fisrt use of dynamic memory.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int* erase_reps(int* array, int array_length, int* final_length) {
5      *final_length = 0;
6      int* result = realloc(NULL, sizeof(*result) * array_length);
7      for (int ii = 0; ii < array_length; ++ii) {
8          int unique = 1;
9          for (int jj = 0; jj < ii; ++jj) {
10             if (array[ii] == array[jj]) {
11                 unique = 0;
12             }
13         }
14         if (unique) {
15             result[*final_length] = array[ii];
16             ++(*final_length);
17         }
18     }
19
20     return realloc(result, *final_length * sizeof(*result));
21 }
22
23
24 int main(void)
25 {
26     int array[] = { 20,1,2,3,4,5,6,5,8,7,9,6,6,5,4,1,2,3,8,5,4,4,5,6 };
27     int length;
28     int* result = erase_reps(array, 24, &length);
29     if (NULL == result) {
30         printf("There's been an error with memory\n");
31         return -1;
32     }
33     for (int ii = 0; ii < length; ++ii) {
34         printf("%d\n", result[ii]);
35     }
36     free(result);
37 }
```

Program 125: Utilización de `realloc`

If you compare both programs, you'd see that in the first version we had to declare an array to have a place to save the the data until we knew the real size. The disadvantage of this is that later we had to copy data to the new zone we were going to return and that we allocated with `malloc` before. In this case we leave the memory manager of the operating system shrink the memory block for us and, being sensible, it is not very probably that when shrinking a memory block it gets moved, although it is possible, so you will save the computational effort to do so in many cases.

The next function is also interesting, `memset` is a function that is declared in header `string.h`. This makes sense because, since it puts every byte to the same value in a memory block, it gets used a lot alongside strings, to fill it with the same letter. The function has this signature:

```c
1  void *memset(void *s, int c, size_t n);
```

Program 126: Signature of function `memset`

The first argument is the pointer where you are going to write the changes, the second is the value we are going to write in **each byte** of the memory zone and the last argument is the number of bytes you're going to write. A very typical example of this is when putting al bytes in memory to zero, this is often done when you reserve memory for some structs whose values you need to initialize or that requieres to be initialized with that value. In later examples we will see them. The example that I am going to show now is a bit more original. Imagine that you want to show a progress bar like this one:

```
[##############################....................]
```

I think you are guessing where I want to get to. To print this bar with a pleasing aesthetic I have to show you a new special character, called carriage return, which is introduce in strings with this sequence: \r. This character takes the printing back to the start of the line, allowing you to overwrite it. Also, I am going to use a function called usleep, that allows us to pause the program for some time, otherwise, the program would make all the priting so quickly we won't be able to see the effect.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5
6  #define ARRAY_SIZE(array) ((sizeof((array)))/(sizeof((array)[0])))
7  #define BAR_LENGTH ((size_t)100)
8
9  void print_bar(int progress) {
10     //all the other chars will be initialized to zero
11     char bar[BAR_LENGTH + 3] = { '[' };
12     //we leave space at the end for the last null char
13     bar[ARRAY_SIZE(bar) - 2] = ']';
14     //we write the punds signs that symbolize the completed part
15     memset(bar + 1, '#', progress);
16     //we write the dots that symbolize the pending part
17     memset(bar + 1 + progress, '.', BAR_LENGTH - progress);
18     //we print and go back to the start of the line
19     printf("%s\r", bar);
20     //this function makes the terminal to update all instantly
21     fflush(NULL);
22  }
23
24  int main(void)
25  {
26     for (int ii = 0; ii <= 100; ++ii) {
27         print_bar(ii); //we wprint the bar
28         usleep((unsigned int)(2.50 * 100000)); //we wait
29     }
30     /*
31       we print a new line to the prompt shows
32       in the next line
33     */
34     printf("\n");
35  }
```

Program 127: Use of the function `memset`

As you can see, the function that prints the bar given a certain percentage is very simple. Due to the fact there are several things I haven't explaing before, I have commented the code exhaustively.

Another useful function is one that allows us to **copy** what is in one memory region to another. This is specially useful because it allows us to copy any kind of data without using a loop, when an algorithm copies several things sometimes the loops that need to be added make the function more complex. Let's see its signature:

```c
void *memcpy(void *dest, const void *src, size_t n);
```

Program 128: Signature of function `memcpy`

It receives three arguments, the first one is the memory zone where you want to copy the data, the second is the memory zone from which we will copy the data and the third argument is the amount of data in bytes you want to copy. To remember which arguments goes first (the destiny or the origin) I use the trick of saying to myself that it works like an assignment, that is, the destiny goes first. Let's see an example, imagine a function that allows u to insert an element in the position we want in an array.

```c
int *insert_at(int *list,
               int  list_size,
               int  position,
               int  element)
{
    int *res = malloc(sizeof(*res) * (list_size + 1));
    memcpy(res, list, sizeof(*res) * position);
    memcpy(res + position, &element, sizeof(*res));
    memcpy(res + position +1,
               list + position,
               sizeof(*res) * (list_size - position));
    return res;
}
```

Program 129: Utilización de la función `memcpy`

The function is very simple, it simply copies from the first position to the position we want to insert the element, then copies the element, which we could do simply with an assignment, and then we copy the elements after the one we wanted to insert.

13.2. Managing strings

Maybe you have noticed that, due to the fact the strings in C are simply char vectors or arrays and that some things are not easy to do with them, managing text is a bit difficult more often than not. If you didn't come to this realization I can tell you that indeed, managing text in C is a bit more difficult than in other languages. But there are several functions to make our life easier. Let's see some of the functionality we have available:

1. Compare texts strings and order them alphabetically.

2. Get the lenght of a string.

3. Duplicate a string.

4. Create new strings with format.

As you can see, we have a very big number of functionality available to us to manipulate strings. These functions are all in the headers `string.h` and `stdio.h` (the same one than `printf`). Let's see function by function.

In general, variables of basic types are compared with the operator ==, but strings in C are pointers. If you compared with this operator, you'd be comparing simply the memory addresses in which both strings are stored, and, unless they're the same, they wouldn't be equal. I guess that with that you know of the language you could write a function that compares to strings and checks if they're equal. Simply write a loop that checks if both strings are equal char by char and checks that the length is the same. The function that does this for us is strcmp.

In general, when you compare things in programming, you invoke an operator or a function that return true if the operands are equal or false if they're different. For example a == b. But in this case, strcmp returns a negative number is the first string goes before in alphabetical order, zero if they are equal, and a possitive number if the first chain goes later in alphabetical order. Let's see an example of its use, let's make a program that indicates if a set of strings in is alphabetical order.

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

int main(int argc, char const* argv[])
{

    if (argc < 3) {
        printf("We need at least two words.\n");
        return EXIT_FAILURE;
    }


    for (int ii = 1; ii < argc - 1; ++ii) {
        if (0 < strcmp(argv[ii], argv[ii+1])) {
            printf("The words introduced as argument are not"
                    " in alphabetical order.\n");
            return EXIT_FAILURE;
        }
    }
    printf("ok\n");
    return EXIT_SUCCESS;
}
```

Program 130: Signature of function strcmp

The conditional is a bit confusing, but it tends to happend when we use strcmp, we want the arguments that have come before to be less than the one we are examining now, so we have to check if the result of performing strcmp is greater than zero (or, as it is written in the code, if zero is less than the result of calling the function) because it would indicate that the word in the next position goes before this one, that is, they're not in order. Also, look at the loop, we start in position one because position zero is the name of the program and we stop before the last one, because inside the loop we check the next position, and there is no next position to the last one.

Next function we're going to look into is strlen, which tells us the length of a string, **without counting the null chacracter**. For starters, before writting the signatrue of the function, let's make the exercise of making our own function that does the same as a simple task:

```
1  size_t my_own_strlen(const char* string)
2  {
3      size_t res  = 0;
4      char   iter = string[res];
5
6      while ('\0' != iter) {
7          ++res;
8          iter = string[res];
9      }
10
11     return res;
12 }
```

Program 131: Propia versión de `strlen`

The function is very simple, but I want you to notice something that is very evident in this function, if the string does not contain any null character, this function would read the next thing in memory without stopping, which, as you already know, tends to make the programs fail and close. Because of this, be careful when using the function, because calling it over a string without a null character would make any program fail.

The signature of the real `strlen` is like this:

```
1  size_t strlen(const char *s);
```

Program 132: Signature of function `strlen`

Briefly, the function that duplicates the strings is `strdup` and its signature is as follows:

```
1  char *strdup(const char *s);
```

Program 133: Definición de `strdup`

It is simply a function to which you pass as an argument the pointer to the string you want to duplicate and return a pointer to the memory block in which it copied the same content. Nevetherless, you must free both zones, if the first one was reserved with dynamic allocation, because this function does not move, it copies.

For example, a common use of this function is when you are creating a struct that stores strings as we saw in the program 61: Uso de punteros constantes como argumentos de función, it us useful that certain structs are managed only with functions. You can come back to the program and change the call to malloc for the string and the copy by a call to `strdup`.

We reach the most important point of this section, the formatting of a string from a set of variables. This is what we have been doing with `printf`. To do this, there are several functions:

1. `sprintf`: Allows to make the same than `printf`, but prints to a char pointer (array or vector) instead than to the terminal.

2. `fprintf`: It is the same idea, but with files, it just receives a `FILE` pointer instead of a `char` pointer.

Again, this is a C manual, and I want it to be continue being more or less strictly that, but is convenient for me to set you an example of the use of these functions. In computer science, sometimes, to transmit information we perform a process called **serialization**. This process is the conversion of data from how it is stored in memory to text format. This is done because different computers may use different ways to store things in memory, for example, there are computers that store byte "backwards". This means that for the number 10.669 which in hexadecimal is: 0x29AD is composed of two bytes (remember, each byte is two hexadecimal digits), in the memory of some computers it would be saved as 0x29 AD and in others it would be 0xAD 29. To avoid this type of confusions, we convert them to text strings.

Coming back to the example in the program 61. We could create a function that creates this serialization of the struct, for example, in the case of a person called José Pérez Martínez, we would like to serialize it like this (again, remember that Spanish names have two surnames):

```
{
    "name":"José",
    "last_name_1":"Pérez",
    "last_name_2":"Martínez"
}
```

Let's see how this function could be written (remember that it must be added to the program I have quoted before) and we will call it `person_to_string`. This function would be like this:

```
1  char* person_to_string(const person_t *p){
2      char preliminar[1024] = {};
3
4      sprintf(preliminar,
5              "{\n"
6              "\t\"name\":\"%s\",\n"
7              "\t\"last_name_1\":\"%s\",\n"
8              "\t\"last_name_2\":\"%s\"\n"
9              "}",
10             p->name, p->last_name_1,
11             p->last_name_2);
12
13     return strdup(preliminar);
14 }
```

Program 134: Basic example of `sprintf`

I have written all the arguments of the function in a different line and I have divided the format (the second argument) in several lines, if you look closely you'd see there is no comma between those lines. Again, remember than string literals separated just by whitespaces are just one. Also, look at how we use an array to format the text because, as it is common already, we do not know how long it is, once we have formated it, we use `strdup` to return a string of the correct size.

Nevertheless, this use of the function is very simple, there is a left out detail I want to mention, that is the value returned by the function. Both `printf` and `sprintf` and all the functions of this family return an integer that is the **number of characters printed** (not counting the null character at the end of the string). This is very useful when you want to print several things in the same string. Let's see an example, imagine a program that, in a similar way, would serialize an array, if the array is 1,2,3,4,5 the serialization would be:

```
[
    1,
    2,
    3,
    4,
    5
]
```

The most immediate solution would be to use `sprintf` to print all the integers, but we have a problem, we do not know how many numbers there are, so we cannot write the format that the function needs. That's why we're going to print on a loop in the return value of the function.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX_STRING_SIZE ((size_t) 65536)
6  #define ARRAY_SIZE(array) ((sizeof((array)))/(sizeof((array)[0])))
7
8  char* integer_array_to_string(const int* array, size_t array_size)
9  {
10     char res[MAX_STRING_SIZE] = {};
11     int  printed_chars        = 0;
12
13     printed_chars += sprintf(res + printed_chars, "[\n");
14     for (size_t ii = 0; ii < array_size; ++ii) {
15         char* separator;
16         if (ii != array_size - 1) {
17             separator = ",\n";
18         }
19         else {
20             separator = "\n";
21         }
22         printed_chars += sprintf(res + printed_chars,
23                                  "\t%d%s", array[ii], separator);
24     }
25     printed_chars += sprintf(res + printed_chars, "]");
26
27     // overflow
28     if(printed_chars >= MAX_STRING_SIZE){
29         return NULL;
30     }
31
32     return strdup(res);
33  }
34
35  int main(void)
36  {
37     int list[] = {1,2,3,4,5,6};
38     char *serialization = integer_array_to_string(list,
39                                                    ARRAY_SIZE(list));
40     printf("%s\n", serialization);
41     free(serialization);
42  }
```

Program 135: Example of an advances use of `sprintf`

If you see how I have written the function, we use thir return value from `sprintf` to concatenate the strings, the mechanism is very simple, if we have printed a certain numbers of characters we must add to the pointer the number of characters printed. Also, we use a conditional inside the loop to avoid printing the last comma in the last element. After the loop, we print the closing bracket and, finally, a conditional checks that we haven't printed more characters that the ones we provisioned for. Notice that we use >= because we have to provision for the last null characters.

Nonetheless, we have a problem, although we can know when the space has been overflown, that is, when we have used more characters than the ones we had, we can't **avoid** this from happening. This has very serious implications, because once you write in a memory zone, you do not know what can happen. Luckily, there is a variation of the function we're using that will help us to avoid such predicament. it is called `sprintf`, and has the same signature that the last one, but it adds an argument: the **maximun** number of character that you want to print, in this way, it will never overflow our memory. Let's see how this could be used in the same program.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAX_STRING_SIZE ((size_t) 65536)
5  #define ARRAY_SIZE(array) ((sizeof((array)))/(sizeof((array)[0])))
6
7  char* integer_array_to_string(const int* array, size_t array_size)
8  {
9      char res[MAX_STRING_SIZE] = {};
10     int  printed_chars        = 0;
11
12     printed_chars += snprintf(res + printed_chars,
13                               MAX_STRING_SIZE,
14                               "[\n");
15     for (size_t ii = 0;
16          ii < array_size && printed_chars < MAX_STRING_SIZE - 2;
17          ++ii)
18     {
19         char* separator;
20         if (ii != array_size - 1) {
21             separator = ",\n";
22         } else {
23             separator = "\n";
24         }
25         printed_chars += snprintf(res + printed_chars,
26                                   MAX_STRING_SIZE - printed_chars,
27                                   "\t%d%s",
28                                   array[ii],
29                                   separator);
30     }
31     printed_chars += snprintf(res + printed_chars,
32                               MAX_STRING_SIZE - printed_chars,
33                               "]");
34     // overflow
35     if(printed_chars >= MAX_STRING_SIZE){
36         return NULL;
37     }
38     return strdup(res);
39  }
40
41  int main(void)
42  {
43      int list[24] = { };
44      char* serialization =
45                  integer_array_to_string(list, ARRAY_SIZE(list));
46      if (serialization != NULL) {
47          printf("%s\n", serialization);
48      }else{
49          printf("Error: límite excedido\n");
50      }
51      free(serialization);
52  }
```

Program 136: Ejemplo de uso de `snprintf`

As you can see, the second argument of the function is the limit of characters that we can keep printing. We calculate this substracting from the length of the array what we have already printed, in that way, if we have alrady printed 100 characters, we have 65536–100 available. On the other hand, take into account that `snprintf` return always the number of characters that **would be** printed regardless of the limit of characters set by the second argument of the function. When printing, the null character at the end of the string counts, that is, if a call to `snprintf` receives as limit three and you tring to print `abc`, it would return 3, but it won't write the last letter, because it always write the null character (unless the limit is zero).

Be careful, you need to check that the argument for the limit is not negative, because it is defined as an unsigned integer, therefore if you pass a negative number, it will a random positive integer. In this code, we check it in the loop condition.

13.2.1. Positional specifiers

In the start of the manual I taught you to print things so you could test your programs, but I did so in a very concise way to avoid consufing you. Nevertheless; I take this chance of coming back to printing functions to develop something I left on the table then. This is positional specifiers, we already know what a specifier is, they indicate, inside the format, which data type has the argument we're going to print in that operation. The problem of this system, which is simple, is that it creates problems when you cant to print something several times in the same printing operation.

If, for instance, we need to print several times the same variable, there is no use in passing it as argument several times. For example, imagine a function that simulates a birth certificate, including the name of the parent, for a father called Fernando García Pérez and a mother called María Fernández López, if the name of the child were Federico, his name would be Federico García López and the thee names would be included in the printing. The function is trivial, let's write it applying what we know up until now.

```
 1  #include <stdbool.h>
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5
 6  typedef struct person_s {
 7      char *name;
 8      char *last_name_1;
 9      char *last_name_2;
10  } person_t;
11
12  char *son_name(const person_t *father, const person_t *mother,
13                 const char *first_name) {
14      const int STRING_LENGHT = 65536;
15      char res[STRING_LENGHT];
16      char *format = "===BIRTH CERTIFICATE===\n"
17                     " - Nombre del Padre: %s %s %s\n"
18                     " - Nombre de la madre: %s %s %s\n"
19                     " - Nombre del hijo: %s %s %s\n";
20      snprintf(res, STRING_LENGHT, format, father->name,
21               father->last_name_1, father->last_name_2, mother->name,
22               mother->last_name_1, mother->last_name_2, first_name,
23               father->last_name_1, mother->last_name_1);
24      return strdup(res);
25  }
26
27  int main(void) {
28      char* text = NULL;
29      person_t father = {"Fernando", "García", "Pérez"};
30      person_t mother = {"María", "Fernández", "López"};
31      text = son_name(&father, &mother, "Federico");
32      printf("%s\n", text);
33      free(text);
34  }
```

Program 137: Example of printing with repeated argument

The function is simple, but we repeat arguments, as we introduced before, this is not only inefficient, but also error prone, because you have to keep count of the position of the arguments. When there are not many, less than five for instance, it is easy, but, the moment the number starts to be bigger than that, getting lost is a real possibility. Hence, telling the function to print in that place the argument in certain position is a nice solution, because it allows us to pass each argument just once to the function.

```c
 1  typedef struct person_s {
 2      char *name;
 3      char *last_name_1;
 4      char *last_name_2;
 5  } person_t;
 6
 7  char *son_name(const person_t *father, const person_t *mother,
 8                 const char *first_name) {
 9      const int STRING_LENGHT = 65536;
10      char res[STRING_LENGHT];
11      char *format = "===PARTIDA DE NACIMIENTO===\n"
12                     " - Nombre del Padre: %s %s %s\n"
13                     " - Nombre de la madre: %s %s %s\n"
14                     " - Nombre del hijo: %s %2$s %5$s\n";
15      snprintf(res, STRING_LENGHT, format, father->name,
16              father->last_name_1, father->last_name_2, mother->name,
17              mother->last_name_1, mother->last_name_2, first_name);
18      return strdup(res);
19  }
20
21  int main(void) {
22      char *text        = NULL;
23      person_t father = {"Fernando", "García", "Pérez"};
24      person_t mother = {"María", "Fernández", "López"};
25      text            = son_name(&father, &mother, "Federico");
26      printf("%s\n", text);
27      free(text);
28  }
```

Program 138: Example of positional specifier

The first change is that now the two last specifiers are special, a positional specifier starts as everyone, with a percentage symbol, the the position, as a number, a dollar sign and the indicator of the data type, in this case an s, because it's a string.

13.3. Error management and manual

Now we have seen several functions and you know how to compile a program I want to come back to an example program from some sections before, the program 79: Example of a program that uses `remove`. If we include the header file `errno.h` we can rewrite the program in this way:

```c
1  #include <errno.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(int argc, char const *argv[]) {
6      if (argc != 2) {
7          printf("Usage: ./main <path to the file>");
8      }
9
10     int error = remove(argv[1]);
11
12     if (error == 0) {
13         return EXIT_SUCCESS;
14     }
15     switch (errno) {
16     case ENOENT:
17         printf("No such file or directory.\n");
18         break;
19     case EACCES:
20         printf("Permission denied\n");
21         break;
22     default:
23         printf("Undetermined error\n");
24         break;
25     }
26     return EXIT_FAILURE;
27 }
```

Program 139: Ejemplo de programa que usa la variable `errno`

As you saw in the program 79 I check the return value of the function I am calling and, if it's zero (what tends to indicate success) I return `EXIT_SUCCESS`, ending the program. Otherwise, if the returned value is not zero, we go into a `switch` about a variable we do not know. In it, we use a series od values that are not present in the program either. This is because we have included the header `errno.h`. This allows us to use the global variable `errno`.

The variable exists do when we call some function that uses it to notify errors, its value would be assigned in consequence to some of the values defined in the header. For example, here we have prepared for some of the cases. Nevetherless; a question arises: how to know which functions use this variable and which don't and what values are present. To do this, the **manual** is used. This is a function of Linux operating systems in which you can call the `man` command to find information about any function or header. For example, try to write in a termina this:

```
$ man errno
```

You would see a result that start with:

```
NAME
       errno - number of last error

SYNOPSIS
       #include <errno.h>

DESCRIPTION
       The <errno.h> header file defines the integer variable errno,
          which is set by system calls and some library functions in
          the event of an error to indicate what went wrong.

   errno
       The  value  in errno is significant only when the return value
          of the call indicated an error (i.e., -1 from most system
          calls; -1 or NULL from most library functions); a function
          that succeeds is
       allowed to change errno.  The value of errno is never set to
          zero by any system call or library function.

       For some system calls and library functions (e.g., getpriority
          (2)), -1 is a valid return on success.  In such cases, a
          successful return can be distinguished from an error return
          by setting  errno
       to zero before the call, and then, if the call returns a status
          that indicates that an error may have occurred, checking to
          see if errno has a nonzero value.
<continúa>
```

In manual screens you move the dispaly with the arrow keys and go out of the manual hitting q. If you want to know the content of any function (functions, variables...) or the signature of any function, just write `man` next to the name of the header or function. If you get results that are not what you were looking for, execute `man 3 <name>`. The manual has information about other things, like linux commands, but section three is the one that contains information about the C functions.

13.4. Exercises of the section

Ex. 18: Rewrite the program of exercise 15 not using the static pointer array (Use `realloc` and `strdup`).

Ex. 19: Write a program that receives an unknown number of words as arguments and shorts them alphabetically and, later, prints them.

Ex. 20: Write a program that receives as argument a word and a number. If the number is zero, it must turn the word into uppercase, it the number is not zero, it must turn it into lowercase.

```
$ ./main.exe Anthony 0
anthony
$ ./main.exe USA 0
usa
$ ./main.exe spqr 1
SPQR
```

Clue: Check the ASCII table to know what distance is there between a lowercase letter and an uppercase letter.

Ex. 21: Make an program that given a number as an argument, prints a pyramid like this one with as many lines as the number indicates:

```
%%%%%%%%%
 %%%%%%%
  %%%%%
   %%%
    %
```

Note: Use `memset`.

Ex. 22: Write a program the receives a series of points and names for each one and then prints them according of their distance to the origin, from nearest to furthest. For example:

```
$ main.exe 2 3 Cincinnati 4 5 Indianapolis -1 3 Birmingham
-1 3 Birmingham
2 3 Cincinnati
4 5 Indianapolis
```

Note: create an struct called `tagged_point` that manages the strings as is seen in the program 62, but using `strdup`.

14.   Advanced logic

Up until now we have been contempt with using any integer type (generally, `int`) to store logic values, but this can be avoided, saving space by creating a datatype that allows us to store properly a logic type. This type is `bool`. To be able to use this type you must include the header `stdbool.h`. Also, this type adds two new keywords: `true` and `false`. Words that symbolize, as you can imagine, a true and a false logical value.

Let's see the use of this type in action:

```
1  #include <stdio.h>
2  #include <stdbool.h>
3
4  int main(void)
5  {
6      bool it_is_difficult = 10;
7      printf("%d\n", it_is_difficult);
8  }
```

Program 140: Example of us of `bool`

There is not a specifier to print booleans, but if you use the `%d` specifier, you'd see that the problem prints "1". That is, if we are rigorous storing our logic values in the boolean type, we can assume that a true logic value is always equal to 1. Nevetherless, we still have a problem, that is the use of a whole byte to store a value that could be stores in a bit. This problem comes from the fact that computers do not memory blocks smaller than a byte. Nevertheless, that doesn't mean we can't manipulate the concrete bits inside a byte, for example, set a bit to zero, invert them... for those tasks we use bit-level operators.

14.1.   Bit-level operators

I am going to be sincere, I didn't know where to put this section, that is, I didn't find a place to put it in the manual. At the start I thought about put it after the logic operators, but the utility of these was difficult to explain at that moment. So it is better to learn them now. There are several bit-level operators, these are:

1. Bit level conjunction, written in math as $\cdot$, it is done with `&`.

2. Bit level disjunction, written in math as $+$, it is done with `|`.

3. Bit level negation, written in math in the same that logic negation, it is done with `~`.

4. Bit shifting to the left, binary operator that moves all the bits of an integer type to the left as many times as the second operand indicates. It is done with the `<<` operator.

5. Bit shifting to the right, binary operator that moaves all the bits of an integer type to the right as many positions as the second operand indicates. It is done with the `>>` operator.

What bit-level means is that, applying these operators to two integer variables, the operator generates another integer whose value would be the result of generating the logic operation that the operator does, but bit by bit. That is, the result of the a conjunction bit by bit is a new integer value in which each bit is the result of the conjunction of the two bits in that position in the operands.

To understand this correctly you need to understand how the binary numbers are represented inside the computer. You already know how the unsigned integers are represented, in binary, that is, as we saw it in section 8.1. Nevetherless; negative numbers are representes un a special way. There are several ways to represent signed numbers in binary, the most naive would be just to reserve a bit, the first generally, for the sign, if the bit is zero, it is a possitive number, if it's a one, it's a negative number. In this way, number 7 would be 00000111 and -7 is 10000111, but the problem of this representation is that we have 0 and -0. And since we're computer programmers, we don't like to waste a valuable number. Because of this, signed numbers are in **two's complement**.

In this system, positive numbers coincide with their natural binary representation, nevertheless, negative numbers are represented with their complement, to get the complement of the number, you need to follow these steps:

1. Invert all the bits (that is, turn zeroes into ones and ones into zeroes).

2. Add one to the result of the last operation.

$$7_{(10} = 00000111_{(2}; \quad \overline{00000111} = 11111000; \quad +\frac{\begin{array}{r}11111000 \\ 1\end{array}}{11111001} \rightarrow -7_{(10} = 11111001_{(2C}$$

The advantage of this representation is that, also, the first bit is just one when the number is negative, so the first bit keeps being an indicator of the sign, as in the naive representation we discussed before. Another advantage of this system is that there is just one zero representation, which is count as a possitive number. This explains why, when we were seeing the range of basic types, integer types had always a possitive range that was one unit smaller than the negative range, the `char`, for example, comes from -128 to 127.

Let's come back to the bit-level operators, let a `char` be -7 and another be 12, let's see how their bit-level conjunction would be calculated:

$$a = -7_{(10} = 11111001_{(CA2} \quad b = 12_{(10} = 00001100_{(CA2} \quad a \cdot b = \cdot\frac{\begin{array}{r}11111001 \\ 00001100\end{array}}{00001000}$$

Shifting is a simple operation, let's choose for example the number 7 again, that is: 00000111, if we apply a left shift of two bit, it would turn into 00011100, in decimal, 28. If you haven't notices it already, I say it now, shifting bits to the left in binary is the same than multiplying the number by two. On the other hand, if we shift bits to the right, it would be: 00000001, the ones that do not fit, are erased, as you can see. This operation, logically, is equivalent to dividing the number by two (in integer division).

The best use for this is that it allows us to stablish something programmers call flags. That is, it allows us to use the individual bits of an integer variable to indicate several logic variables. For example, we can create a function that serializes an object of type person, with these options:

1. Serialize with legible names, that is: "first last name" instead of, for example, "last_name_1".

2. Serialize with spaces or without them betwen control characters, that is, `"last_name_1" : "Johnson"` instead of `"last_name_1":"Johnson"`.

3. Serialize in several lines.

In general, if these options **were exclusive**, they couldn't be together, we could codify them simply as an enumerated type, but since they're not, they must be different logical variables, the problem with this is that it would force us to pass three boolean arguments to the function. To avoid this, we're going to use the so-called flags, we are going to assign a bit to each option, and we are going to create an anumerated type where each option has the value of an integer with that bit set to one. To pass several options, the user can simply perform a bit-level disjunction with the options. Let's see how this would be written.

```c
1  #include <stdbool.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  typedef enum {
7      LEGIBLE_NAMES   = 1 << 0,
8      CONTROL_ESPACES = 1 << 1,
9      MULTIPLE_LINES  = 1 << 2
10 } serialization_options_t;
11
12 typedef struct person_s {
13     char *name;
14     char *last_name;
15     unsigned int age;
16 } person_t;
17
18 char *serialize_person(const person_t *person,
19                        serialization_options_t opts) {
20     char *field_names_legible[] = {"Name", "Surname", "Age"};
21     char *field_names_normal[]  = {"name", "last_name", "age"};
22     char **field_names          = NULL;
23     char *separator             = NULL;
24     char *line_end              = NULL;
25
26     bool legible   = opts & LEGIBLE_NAMES;
27     bool espaces   = opts & CONTROL_ESPACES;
28     bool multiline = opts & MULTIPLE_LINES;
29
30     field_names = legible ? field_names_legible : field_names_normal;
31     separator   = espaces ? " : " : ":";
32     line_end    = multiline ? "\n" : "";
33
34     char *fmt = "{%7$s\"%1$s\"%8$s\"%4$s\",%7$s"
35                 "\"%2$s\"%8$s\"%5$s\",%7$s"
36                 "\"%3$s\"%8$s%6$u%7$s}";
37
38     char res[65536];
39     sprintf(res, fmt, field_names[0], field_names[1], field_names[2],
40             person->name, person->last_name, person->age, line_end,
41             separator);
42     return strdup(res);
43 }
44
45 int main(void) {
46     person_t myself = {"Francisco", "Rodríguez", 26};
47     char *text = serialize_person(&myself, MULTIPLE_LINES |
48                                   LEGIBLE_NAMES | CONTROL_ESPACES);
49     printf("%s\n", text);
50     free(text);
51 }
```

Program 141: Implementation of flags with bit-level operations

The enumerated in line 8 is a bit intimidating, but if you remember when we explained them, an enumerated type can be defined with custom values for each symbol, here we use the left shift operator to create values that have only one bit set to one, in different positions. I could have assign them the values 1, 2, 4, but with the shifting I can see the flag value better because I know it must be `1 <<` and then a consecutive number, I don't need to keep multiplying by two.

After the definition of the struct person, which has nothing special. In line 20 we reach the function to serialize and, as is logical, it receives a constant pointer to the struct and the options. At the start we declare variables, nothing uncommon. The interesting thing is in line 28, I am calculating boolean variables from the flags that come from the last arguments, for each one of the flags. The way is very simple, when you make a conjunction with a flag value and the actual options passed as an argument, it will be true if the flag was up in the argument, false otherwise. Let's see the multiline option as an example. If `MULTIPLE_LINES = 1 << 2`, that is, 00000100, and the options are, for example, 00000101, performing the conjunction it would be, 00000100, that is a value different from zero and therefore, true.

In the next line I use a language artifact called ternary operator, which is useful when you have several operations based in logical values. It is called ternary because it takes **three** operands, its syntax is like it follows: `condition ? value1 : value2`, the operator would return `value1` when the condition is true, and `value2` when the condition is false. Be careful because both values must have the same type. I use it here to define the separators of field, line and the names for the serialization.

Finally, in function `main` we call the serialization function with the options, simply we chain them all together with a bit-level disjunction. Be careful not to misspell when you use uptions and use wrongly the logic operator.

On the other han, there is an operation I'd like to talk you about, that is when we need to pull down a flag inside a flag group. Imagine we wanted to serialize a struct with all the options but immediately after one without one of them, for example, we want to pull down the multiline flag, we could create the options in both calls, but, if we wanted, we could save the options in a variable, use them and then pull that flag down. To do so you must make a conjunction with the negation of the flag value. Let's see how it would look.

```c
int main(void) {
    person_t myself = {"Francisco", "Rodríguez", 26};
    char *text      = NULL;

    serialization_options_t opts =
        LEGIBLE_NAMES | CONTROL_ESPACES | MULTIPLE_LINES;

    text = serialize_person(&myself, opts);
    printf("%s\n", text);
    free(text);

    opts = opts & ~CONTROL_ESPACES;

    text = serialize_person(&myself, opts);
    printf("%s\n", text);
    free(text);
}
```

Program 142: Example of pulling down a flag

This is a bit counterintuitive, let's see it with the values in the example, in line 12, `opts` is 00000111, optione `CONTROL_SPACES` is 00000010, if we invert or negate it, we get 11111101, if you make the conjunction bit by bit with that value, you'd see that all of them would remain as they were before, but the one corresponding to the flag, that one would be forced to become zero. That's how you pull down a flag in a series of options. Be aware that, for all bit-level operation there are the operators that combine them with assignment, that is: `|=`, `&=`, `~=`, `<<=` and `>>=`, so we could rewrite the line we were talking about as: `opts &= ~CONTROL_ESPACES;`.

15.   Algorithms

An algorithm, as we explained in the introduction, it the set of steps you must follow to reach a goal. The programs we have done have a series of goals, that are fulfilled using algorithms. In this section, I want to explain the first concepts about them and, specially, present some that would allow you to interiorize some code patterns. Also, in later sections we could use these simple algorithms to introduce more complex concepts.

The first thing we're going to do is see how to express an algorithm, an algorithm can be explained sometimes in normal language, that is, as humans talk, if we already have code that executes it, we are also expressing the algorithm, but sometimes it is needed to use middle ground tools, firstly: because it could be difficult to codify the algorithm directly without thinking about it before and, secondly: because doing this allows us to think about it without thinking about the concrete artifacts of the language we are going to use, which allows us to leave that work for later on.

These ways of expressing them are plentyful, for example, flow diagrams as I used them to explain how loops worked are one way. Another is the **pseudocode**, this is a concept that allows us to express algorithms in a structured way, using basic artifacts of any programming language but in a more relaxed way. Let's see it with an example: the algorithm that allos us to delete the repetitions in an array as explained in the program 56: Example of dynamic allocation.

```
algorithm erase_reps :=
input: array
solution = {}
for ii from 0 to size(array) − 1:
    element = array[ii]
    unique = TRUE
    for jj from 0 to ii − 1:
        element2 = array[jj]
        if element equals element2:
            unique = FALSE
    if unique
        add element to solution
return solution
```

As you can see, it is more easy to understand, because we are getting rid of several aspects of the language we're not interested in now, for example: assume we can know the size of the array without the need to worry about where it comes from; we do not have to convert the logic variable to numbers; we can assume that adding an element to the array is self-explanatory; we can ignore dynamic memory allocation and simply say we return an array. Doing this after you already have the code is useless, but think about how useful this could have helped.

The problem of pseudocode is that we can "cheat", that is, we can always ignore some important things that, when trying to write it in real code, are not obvious. Because of this, you can dedice how much of it you ignore or include. In this case, for example, we could have included the fact that the dimension of the array can't be known inside a function *per se*.

15.1.   Recursivity

When defining some algorithms, they are defined used themselves, that is, part of themselves includes their own use. A classic example of this is the factorial number. Being n a natural number, the factorial of n is written as $n!$, which is equal to multiplying the all the numbers from 1 to n, that is:

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot 4 \ldots n$$

Continuing with the pseudocode, the most evident way to define this is the **iterative**, that is, the one that uses repetitions with loops, iterative solutions tend to be opposed to recursive definitions. Let's see the pseudocode for this way of solving the problem:

---

algorithm factorial :=
input: n
result = 1
for i ***from*** 1 ***to*** n:
    result = result*i
return result

Nevetherless; we can simply define the factorial function using itself, let's go back to its mathematical definition:

$$n! = \prod_{i=1}^{n} i = n \cdot \prod_{i=1}^{n-1} i = n\,(n-1)!$$

Therefore, if we continue with this reasoning:

---

algorithm factorial :=
input: n
if n ***equals*** 0:
    return 1
return factorial(n − 1) * n

If we jump right to the implementation, we will see that is is as easy as the function calling itself. As you very well know, a function can call another, and so on, but nothing avoids us to call the very function we're executing right now. The only problem this presents is that, if it is poorly designed, you can make the function to call itself indefinitely, this would make that, since each non-finished call adds things to the stack, this would end up filling up totally and the program will fail. To see this comparison in the programs with both solutions, we will create both versions of this function, the iterative one and the recursive, the first one would be called `factorial\_interative` and the second is `factorial_recursive`.

```
1  unsigned long factorial_recursive(int n)
2  {
3      if (n == 0) {
4          return 1;
5      }
6      return factorial_recursive(n - 1) * n;
7  }
8
9  unsigned long factorial_iterative(int n)
10 {
11     if (n == 0) {
12         return 1;
13     }
14     unsigned long res = 1;
15     for (int i = 1; i <= n; ++i) {
16         res *= i;
17     }
18     return res;
19 }
```

Program 143: Comparison of iterative and recursive algorithms

This example is classic, and it is so because you can see very well the differences between the implementations and, also, the parts of a recursive function. A recursive function, in general terms, it must have two components: the so-called base case and the recursive call (or calls). The earlier is here represented by the conditional, a recursive function is so because it is defined in a circular way, that is, it calls itself, to be able to go out of that cycle, you need to have a moment in which the function has an evident solution. In this case is when n is equal to zero, case in which the answer is 1, because that's the mathematical definition. The recursive call is in the next line.

Comparing the two solutions, the first one if what is called "more elegant", in the sense that it is more legible (so much one could say it is trivial) while the iterative is longer and less obvious. Nevetherless; the iterative solutions are more efficient than the recursive. This is because the several calls to the function add up into the stack, and they have a cost, each time you call this function, n will be copied into the stack, until you reach the call that corresponds to the base case. In that moment, the call to the base case will return and this succession of calls will unroll itself.

On the other hand, the iterative solucion has no need to copy anything, nor keep calling the functions again and again, it is a simple loop. Is it reasonable, therefore, to think that the first solution will be slower. Let me to check it and I will come back with the results so you can see it. The recursive solution is slower than the iterative, being 1.7431 times slower, therefore it is noticeable worse.Also, remember the stack, it uses more memory. To sum it up: iterative algorithms thend to be faster than their recursive counterparts, but less elegant.

Another classic example to appreciate the differences between both ways to write a function can be seen in the succession of the Fibonacci series. Shallowly: this succession was created by an Italian mathematician in the XIII century, whose name was given to his succession. It is defined in this way: the two first elements of it are 1, and the others are defined as the addition of the two last terms (you can see here the base case and the recursive definition). Mathematically, it would be defined in this way:

$$a_1 = 1, a_2 = 1, \forall_{n=3}^{\infty} (a_n = a_{n-1} + a_{n-2})$$

I think you understand that, this could be written iteratively with a loop, in a similar way to the factorial. So we will jump right to the recursive definition.

```
1  unsigned long fibonacci(unsigned int n) {
2      if (n < 3) {
3          return 1;
4      }
5      return fibonacci(n - 1) + fibonacci(n - 2);
6  }
```

Program 144: Function to calculate Fibonacci succession

As you can see, we follow the same pattern: a conditional that checks if we're in the base case and a recursive call. The base case is simple, the first and second element are one, therefore this is the value we must return if n is less than 3. The recursive call is the one we are interested in, as you can see, there are two recursive calls, for each call that is not the base case. This is important, because you must have a clear idea of in which order these calls are made. Hence I am going to represent it here:

- We call the function for the value 5.
  - The function is called for the value 4.
    * The function is called for the value 3.
      · The function is called for the value 2.
      · 1 is returned.
      · The function is called for the value 1.
      · 1 is returned.
    * 2 is returned.
    * The function is called for the value 2.
    * 1 is returned.
  - 3 is returned.
  - The function is called for the value 3.
    * The function is called for the value 2.
    * 1 is returned.
    * The function is called for the value 1.
    * 1 is returned.
  - 2 is returned.
- 5 is returned.

These nested lists would represent how the sequence of calls to the recursive functions. You can see several things, the firt one being that even for a small value of n, five, a high number of calls are performed. On the other hand, a lot of the calls are done more than once, the call for three is repeated three times, the call for two, three times... and this is only when the original n is five. This gives you an idea that the recursive solution for this is very inefficient. More advanced thechniques allow to solve this. To sum it up, this way of writting this is not very good.

And, if this is the case, if recursive algorithms are so inefficient, the logical conclusion would be to think they're not used. On the contrary, they are used a great deal, because there are algorithms that are defined just recursively, that is, they're not an iterative algorithm we have defined with a recursion. Those algorithms would be difficult to show now and won't leave clear the basic structure of the recursive algorithms.

15.2. Sorting algorithms

Una of the most basic kinds of algorithms are the sorting algorithms, among them we will see the following ones:

1. Bubble Sort.

2. Selection.

3. Insertion.

4. *Quick sort*.

This is like the case because this task (sorting a vector or array) is very common and allows us to deploy a big amount of innovation, many authors of scientifical works are putting sfforts still in improving these algorithms. Also, it is an operation very common in computing. The first algorithms are more or less simple, so we will speak directly over the C implementation, but the last two will need a more thorough explanation.

The bubble sort algorithm is based upon comparing each element with the next one and, if they're not in the correct order, swap them. Let's see how this could be done.

```c
void bubble_sort(int *list, int list_size) {
    for (int ii = 0; ii < list_size - 1; ++ii) {
        for (int jj = 0; jj < list_size - 1; ++jj) {
            if (list[jj] > list[jj + 1]) {
                int aux = list[jj];
                list[jj] = list[jj + 1];
                list[jj + 1] = aux;
            }
        }
    }
}
```

Program 145: Bubble sort implementation

This is the simplest sorting algorithm because, as you can see, it is made out of just two nested loops and an a conditional. Quickly: you iterate over the array, but since you're checking with the next element, you stop one position before the end. If the first element (`list[jj]`) is greater than the one that comes after, you swap them. To swap them you simply sabe one in an auxiliary variable, you assign the second to the first, and the the auxiliary variable to the second. This, done only once, would make the greater element to float up to the last position. And I have said "float" because this is where the name of the algorithm comes from, because each element goes to the highest position as a bubble in water. As a matter of fact, you can try and execute the program after changing the condition of the first loop to `ii < 1`. You will see how only the last element is the one ordered.

This algorithm has a very clear advantage: it uses very little memory. More so, if we ignore the auxiliary variables for the loops, the stack arguments and such, the only occupied memory is the auxiliary varuable, that is, the size of the datatype that you're ordering, in this case: four bytes. In computer science there is always a trade-off between memory and speed. Algorithms that use much memory to do the same tend to be faster than the ones that use very little memory. Bubble sort is a good example of this.

As a final conclusion, we're going to pay some attention to the realization of the swapping: since it is something I am going to use in several of the algorithms, we're going to create a function called `swap`, that allows us to swap two elements of integer type. Let's see how the algorithm would end up with this function defined.

```c
void swap(int *a, int *b)
{
    int aux = *b;
    *b = *a;
    *a = aux;
}

void bubble_sort(int* list, int list_size)
{
    for (int ii = 0; ii < list_size - 1; ++ii) {
        for (int jj = 0; jj < list_size - 1; ++jj) {
            if (list[jj] > list[jj + 1]) {
                swap(&list[jj], &list[jj + 1]);
            }
        }
    }
}
```

Program 146: Implementation of `swap` and use in bubble sort

The next algorithm is selection. This algorithm looks for (selects) the smallest element in the array and puts it in the last position of the array we have ordered. At the begining there is none, when you have put one, you must put the next element in the position next to that one, and so on, let's see it:

```c
void selection_sort(int *list, int list_size)
{
    int ordered = 0;
    while (ordered < list_size) {
        int min_value = list[ordered], min_pos = ordered;
        for (int jj = ordered; jj < list_size; ++jj) {
            if(list[jj] < min_value){
                min_value = list[jj];
                min_pos = jj;
            }
        }
        swap(&list[ordered], &list[min_pos]);
        ++ordered;
    }
}
```

Program 147: Algoritmo de selección

If you think about it, this algorithm uses slightly more memory that the bubble sort. In this case we must maintain always in memory the value and the position of the smallest element. On the other hand, it is a bit faster that bubble sort, why? Because, in the worst case scenario, bubble sort performs $\frac{n(n-1)}{2}$ comparisons (n being the length of the array), the same than this one and, nevertheless, performs much more swaps, because elements go "step by step" to their position, instead of directly like here. Hence in the worst case scenario, this algorithm would perform just $n-1$ swaps, while bubble sort may need as many as comparisons (because each comparison would execute a swap).

The next algorithm is the insertion one. In the last one, we choose the smallest element and we put it at the start, in this one, the reasoning is the opposite, we create a list that we will define as ordered, initially empty, and later we will pick an element from the original list and insert it ordered in the correct position in the sorted list. In this way, the ordered list will always stay sorted. Let's see some pseudocode:

```
algorithm insertion_sort :=
input: array
sorted_list = {}
for i from 0 to size(array):
    insert_sorted(sorted_list, array[ii])
array = sorted_list
```

The problem if this pseudocode is that inserting in an ordered list is something complex enough that will need its own definition. Let's define the algorithm to insert an element in a list, but that **is already allocated in memory**.

```
algorithm insertar_ordenado :=
input: array, element
inserted = 0
for i from 0 to size(array):
    if array[ii] is greater than element:
        insert(array, element, ii)
        inserted = 1
        break
if not inserted:
    insert(array, element, ii)
```

But we still have the problem of inserting, in general, an element in a given position, therefore, we will use another algorithm.

```
algorithm insert :=
input: array, element, position
inserted = 0
for i from size(array) to 0 stepping in −1:
    array[ii] = array[ii − 1]
array[position] = element
```

It is a bit confusing, be what we do is, firt, move the elements after the position in which we want to insert the new element one position right, opening the space for the new element, after that, we insert the element. As you can see, in this algorithm we have to use some intermediate functions. Although this overall sorting algorithm may be implemented in a simpler way, we're going to do it like this so you can see the real power of using functions that allow you to split the tasks in smaller ones. The implementation of the functions would be as follows.

```c
1  void insert_at(int* list, int list_size, int position, int element)
2  {
3      for(int ii = list_size; ii != position; --ii){
4          list[ii] = list[ii-1];
5      }
6      list[position] = element;
7  }
8
9  void insert_at_ordered(int* list, int list_size, int element) {
10     int inserted = 0;
11     for(int ii = 0; ii < list_size; ++ii){
12         if(list[ii] > element){
13             insert_at(list, list_size, ii, element);
14             inserted = 1;
15             break;
16         }
17     }
18     if(!inserted){
19         insert_at(list, list_size, list_size, element);
20     }
21 }
```

Program 148: Auxiliary algorithms for insertion sort

```c
1  void insertion_sort(int* list, int list_size)
2  {
3      int ordered_list[list_size];
4      int ordered_size =0;
5      for (int ii = 0; ii < list_size; ++ii) {
6          insert_at_ordered(ordered_list, ordered_size, list[ii]);
7          ordered_size++;
8      }
9      memcpy(list, ordered_list, sizeof(int) * list_size);
10 }
```

Program 149: Insert algorithm

This algorithm is the one we tend to use unconsciously to sort physical objects when we have them. If you had to sort the numbers $\{5, 4, 6, 9, 8\}$, you'd probably get the five and put it after four, then you'd see six is in order then you'll see nine and put it after eight. Anyways, if you have thought that you'd do selection, yes, it could be possible some people sort things in that way too.

The way we have implemente it is very ingenous, we use many more steps than needed. There is a simpler way if we realize that we can do all the operations in the same list. To give you an example, imagine the list: | **2** | **6** | 2 | 5 | 7 | Cells in boldtext are the elements of the sorted list, what we are going to do is to save the next element after them in a variable, that is, two. After that, we're going to move all the elements from the sorted list greater than two a position right, leaving the array in this way: | **2** | | **6** | 5 | 7 |. With an empty position, as you can see. This cell is used to put in place the element that was saved in a variable as we said before, ending the array like this: | **2** | **2** | **6** | 5 | 7 | If you continue by hand this execution you'd see that we will save the five, we will move six, we will put five on the left of the six; we would sabe seven and we will move the six back, we could put seven in the free position and the array would end up ordered. The implementation of this would be as follows:

```
1  void insertion_sort_optimized(int *array, int array_size)
2  {
3      int element;
4      for (int ii = 1; ii < array_size; ii++) {
5          element = array[ii];
6          int jj;
7          for (jj = ii - 1; jj >= 0; --jj) {
8              if (array[jj] <= element) {
9                  break;
10             }
11             array[jj + 1] = array[jj];
12         }
13         array[jj + 1] = element;
14     }
15 }
```

Program 150: Alternative implementation of insertion algorithm

The logic of this is perhaps less evident. The outer loop os very simple, we iterate over all the array **from the second position**. We save the the next element to the sorted ones in the variable `element`. Aftet that, in this inner `for` loop we will move all the elements of the sorted list to the right one position, doing this is easier backwards, that's why `jj` is decremented instead of incremented as has been usual. When we have found the first element that is greater, we stop (`break`). Finally, we put the saved element on its place.

Finally, we reach the Quick sort algorithm, whose name, as you can see, is not descriptive of how it sorts, but of a nice quality it has: it is very fast. If you remember what I said a few paragraphs before, in computing, speed and memory are fighting metrics, that is, when you favour one, the other gets worse. I am very interested that you see this algorithm because it is **a recursive algorithm**. And it is one of those that is purely this way. Also, since it's more comple, as you may have guessed, let's see its pseudocode first.

```
akgorithm quick_sort :=
input: array
little_list = {}
big_list = {}
pivot = array[0]
if size(array) equals 1:
    res = {pivot}
    return res
for i from 0 to size(array):
    if array[i] is less than pivot:
        add(array[i], little_list)
    else:
        add(array[i], big_list)
little_list_sorted = quick_sort(little_list)
big_list_sorted = quick_sort(big_list)
res = join(little_list_sorted, pivot, big_list_sorted)
return res
```

This algorithm can be summed up in the following: the base case is when the list has only one element, because a list with only one element is always sorted. When we're not in the base case, we create two lists: the elements smaller than the pivot will go in one, and the ones equal or bigger will go in the other. The pivot is simply any element of the list we want to sort. At this level it is not relevant, even when its selection affect very much edge cases, that is, with concrete arrays. After making both lists, we will call Quick Sort on both lists, and we will save both sorted lists returned by the algorithm. The result would be the concatenation of the sorted list of smaller elements, the pivot and the sorted list of bigger elements. The implementation of this algorithm would be as follows:

```
1  int *quick_sort(int *list, int list_size)
2  {
3      int *biggers          = NULL, *smallers         = NULL,
4          *biggers_ordered  = NULL, *smallers_ordered = NULL,
5          *res              = malloc(list_size * sizeof(int));
6
7      int bigger_size = 0, smaller_size = 0, pivot = 0;
8
9      if (list_size == 1) {
10         *res = list[0];
11         return res;
12     }
13
14     if(list_size == 0){
15         return NULL;
16     }
17
18     pivot = list[0];
19     biggers = malloc(sizeof(int) * list_size);
20     smallers = malloc(sizeof(int) * list_size);
21
22     for (int ii = 1; ii < list_size; ++ii) {
23         if (list[ii] < pivot) {
24             smallers[smaller_size] = list[ii];
25             smaller_size++;
26
27         }
28         else {
29             biggers[bigger_size] = list[ii];
30             bigger_size++;
31         }
32     }
33
34     biggers  = realloc(biggers, sizeof(int) * bigger_size);
35     smallers = realloc(smallers, sizeof(int) * smaller_size);
36
37     biggers_ordered  = quick_sort(biggers, bigger_size);
38     smallers_ordered = quick_sort(smallers, smaller_size);
39
40     memcpy(res, smallers_ordered, sizeof(int) * smaller_size);
41     res[smaller_size] = pivot;
42     memcpy(res + smaller_size + 1,
43                biggers_ordered, sizeof(int) * bigger_size);
44
45     free(biggers);
46     free(smallers);
47     free(smallers_ordered);
48     free(biggers_ordered);
49     return res;
50 }
```

Program 151: Implementation of Quick Sort

The algorithm is a bit more complex in this way that it was in pseoudocode, but you must follow parallelly to it. The first thing we do declare all the variables. Again, we are going to declare them together, otherwise functions would be very long. Notice that we initialize without checking anything about `res` (which is where we save the result) calling `malloc` to allocate a vector of the same size of the list. There is not danger in calling `malloc` with a size that equals zero, it is safe. After that, as is usual in our recursive functions, we check if we're or not in the base case, the base case is a list with zero element or a list with one element. Both are sorted by definition.

After this, the recursive case starts, the first thing is to read the value of the pivot, we do it now because we're not ensured the list has at least one element. Now we allocate memory for the lists of the bigger elements and the smaller elements. We have already seen this before: we do not know how long they would be, but it is obvious that neither of these lists can be longer than the original ones. After this we go inside a loop that will sift the the elements that go in any of the lists. Since we're limited here by C, we need to do this in this way: we copy the first element of the positions next to the end of the list (which is equal to the size of a list) and we increment in one the size. Note that this loop starts in one, this is to avoid copying the pivot, because if we did evaluate it in the loop, we will be duplicating it. Once we have finished that loop, we redimensionate the lists to their real size. In the same wat than with `malloc`, if you call `realloc` with a zero size, there is not problem, the pointer returned is still valid to be passed to `free`, and you need to do so.

Now we have both lists, we simply call the same Quick Sort that will return us the sorted lists. In the same that when we learnt how to use `memcpy`, we copy together the list of smaller elements, the pivot and the list of bigger elements.

Now we have seen the conceptual algorithm and the implementation of several sorting algorithms, we're going to compare their performance in terms of execution time. What I am going to so is to create a vector a big size, I am going to sort it with an algorithm, I will introduce random data to the vector, I will sort it again and so on for a number of times. The results are, for 262,144 elements:

Algorithm	Time (s)
Bubble sort	243.0596
Selection	71.2807
Insertion	56.7917
Insertion (optimized)	37.1698
Quick Sort	0.0656

Table 11: Execution time of the different sorting algorithms

As you can see, in each algorithm there is an improvement from the other ones, but it is striking that Quick Sort is so much ahead of the others. As a thought: I wouldn't call any algorithm in such a way if it weren't true. Nevetherless, there are some caveats you need to know about these algorithm beyond their speed. For example, as we said, Quick Sort uses much more memory than all the others. If we used this algorithm in certain machined we may have been limited by memory. Also, there is a quality of algorithms called **stability**, which represents how constant the use of memory, time or both things is when the data introduced changes. In the case of these algorithms, the only two that are stable are bubble sort and selection. This means that results of the other ones can change a lot in certain cases, hence of working in environments where these cases happen frequently or where stability is more important that average speed, maybe you should use one of the slower algorithms.

The most evident example of this is the last algorithm, which behaves poorly in the worst case scenario. For Quick Sort, the worse scenario is, ironically, an already ordered array (or inversely ordered). Is you come back to its description or implementation, you'd see why easily. If the array is ordered, all the elements would follow consistently in either the bigger elements list or the smaller elements list, because we use the first element as our pivot. This means that each level of recursivity we will reduce the problem in just one element. This implies that, in this case, for a vector of a thousand elements, we will perform a thousand recursive calls that will make as many copies of the vector with sizes 999, 998, and so on. This is not sustainable with any sizeable vector. Let's compare the selection algorithm (which is stable) with Quick Sort, with an already sorted vector.

The first fact that strikes to when reading the table is that I can't use the same length of the vector that in the last tests, this is because my computer has not enough memory so Quick Sort finish correctly under these circumstances. I have used a vector with 65,536 elements. The selection algorithm takes 4.58 seconds and Quick Sort takes, no more or less than 21.55. Here is where stability, selection is en average worse, but is never much slower than that. As a matter of fact, since I can execute selection over 262,144, let's see how much it takes and compare it with the random vector.

Selection takes 72.05 seconds, as you can see, almost exactly the same that with the random vector. Also, the memory consumption is always the same, quality it shares with the algorithms that were explained before it. This shows that there are things you must take into account apart from the average execution time of an algorithm. All those factors would be object of a book by themselves, and of an algorithms course which is not what we're doing here, the reason of this section is enunciate the existence of these kinds of problems and show the importance of the election and implementation os algorithms for tasks even as simple as sorting an array.

15.3. Search

Searching is another of the most important things a programmer uses, but search depends a lot on where you're searching. Up until now we only know arrays or vectors, which regarding searching, are the same. Searching is, given a value, finding in the structure you're searching in the same value, or its first occurrence. It is evident how to solve this problem in general, a loop that checks if the concrete element it equal to the element we're searching for. If this is the case, you can return the index you have found the element in, otherwise, a value to indicate it hasn't been found is returned, generally -1 or any negative value.

But there is a way to search faster in the array or vector, if this is sorted, you can perform binary seach. It is called binary because you move in both directions: you start in the center of the array and check is this element is greater or less than the element you're looking for, if it is greater, you go to the left part of the array, and repeat the search there, dividing the number of elements you're searching among. Let's see the pseudocode:

```
algorithm binary_search :=
input: array, target

while array not equals {}:
    pos = center(array)
    element = array[pos]
    if element equals target:
        return pos
    if element is less than target:
        array = right_half(array, pos)
    if element is greater than target:
        array = left_half(array, pos)

return −1
```

Basically, we take advantage from the fact that the array or vector ir sorted so, when we choose a position, we know that, necessarily the element we choose must be either in the left side or on right side of the array. We divide by two the size of the problem in each iteration, until there is an element that is equal to our target, or none, therefore the element is not present in the vector. Let's see the implementation:

```c
int binary_search(int* array, int array_size, int target)
{
    int low_end = 0;
    int high_end = array_size;
    while (high_end != low_end) {
        int pos = (high_end - low_end) / 2 + low_end;
        int element = array[pos];
        if (element == target) {
            return pos;
        }
        else if (element < target) {
            low_end = pos;
        }
        else if (element > target) {
            high_end = pos;
        }
    }
    return -1;
}
```

Program 152: Binary seach algorithm implementation

Since dividing array in C is complicated, let's play with the positions. We define a lower and an upper bound, which will be the limits of the part of the array we're looking for the element. What we do is to set ourselves in each iteration in the middle zone of the array and check is we have found the element, if not, we check if it's greater than or less than the target. If the target is greater than this point in the array, we know that our element, if present, is in the right size, that is, later positions, otherwise, we know it is in the left size, in the earlier positions. If we reach the situation in which the limits collide (that is, they are the same), we have ruled out all the possible positions and, therefore, the element is not present in the array.

The advantage of this algorithm resides precisely in something I have gone through quickly some paragraphs before, each time we check if an element is the one we want, we divide the problem by two. This means that, on average, we will make $\log_2(n)$ checks. In the case of the "normal" search, on average, we will make $\frac{n}{2}$ checks. This makes this algorithm more desirable, although we must take into account that the array must be sorted and, if it is not, we must order it, which has a computational cost. Ideally this algorithm is used with structures that are always sorted, that meand, where insertions are performed ordered, but this has a cost because you need to move elements in the insertion operation (in a similar fashion to what we did in the selection sorting algorithm).

16. Generic algorithms

Now we already know what an algorithm is in its most fundamental terms and that we know some of those that are complex to some degree, we can take one step further: make those algorithms "purer", made them in such a way that they are just a succession of steps to calculate something, but they ignore the data type they're operating on. This being said, it's a bit confusing, but you're going to understand it easily. In the last section we have implemented several algorithms that work with arrays and, even when I didn't say it, if you come back now and look at the example code, you will see they're all integer vectors, not of any other type. The problem is that, as you may guess, the process for sorting an array of integers and an array is similar, or even equal, to sorting any data type that supports to be ordered. For example, floating point numbers.

You may think that we can always copy the code to sort integers and substitute `int` for `double`. We could, but this way of proceeding has a very big caveat: it duplicated code. Duplicate code is bad becase of two reasons: it makes our executables bigger, and our code in general **less maintainable**. Put in a simple way, it is more difficult to send such code to anybody so they can understand it easily and, in the case of some error or not defined behaviour, we had to find every version of the functions or piece of code with the problem and rewrite it as many times as it is duplicated to solve the error. Furthermore, we're programmers, our drive is to do more with less.

To do this we have generic tools, the first one being the pointer to `void`. When I presented it `NULL` to you alongside the dynamic memory allocation, I told you that `malloc` return a `void` pointer that will be converted implicitly to any other pointer type. Also, later I told you about `realloc` and `free` they receive `void` pointers that you do not need to convert to such type to pass. To sum it up: in C the `void` pointer has an implicit conversion to all the types, and all the type to `void`.

This is very powerful, because we can receive a `void` pointer when we do not know that to receive. The problem is that a pointer of this type **cant' be dereferenced**. As I said before, you cannot interpret a pointer to `void`. You can only decode it if you convert it to a type first. For example, let's write a function that inverts an array **of any type**.

```
1  void invert_array(void* array, int array_size, int type_size)
2  {
3      void *var = malloc(type_size);
4      for (int ii = 0; ii < array_size / 2; ++ii) {
5          void* element = array + (ii * type_size);
6          void* opposite = array + (array_size - 1 - ii) * type_size;
7          memcpy(var, element, type_size);
8          memcpy(element, opposite, type_size);
9          memcpy(opposite, var, type_size);
10     }
11     free(var);
12 }
```

Program 153: Inversion of an array of any type

As you can see, it is very similar to that you would do if you knew the type, but you need to receive as an argument the size of the data type. What we do in the loop if to create two pointers that represent the elements we need to swap, we can't do it with variables because we do not know the type. These pointers are not needed, but being sincere, without them the lines would be very difficult to read. After that, we use `memcpy` to being able to move the element in position `ii` to an auxiliary variable, and the element in the opposite position (`ii - 1 - array_size`) to the ith position and later the variable to the opposite element.

The first thing to notice is very obvious: a functions that was very simple becomes somewhat more difficult to read, because we cannot use operator, but we have to use function calls. You must admit that, now, we can reverse all the kind of array without suplicating code. There is also an important matter here, while the assignment operator and know the type allow the compiler to map the operations you write with operations inside your processor to move bytes in packs of four or eight, for example. Doing this like this, with `memcpy`, you force the computer to copy byte by byte. This makes generic algorithms written in this way a bit slower than their specific counterparts.

There are several examples of algorithms that could use well this technique, but with only this tool we cannot solve the problem completely. We are going to create a function that prints an array of generic elements. The function, with the tools we have, would look more or less like this.

```
1  void print_array_generic(void*       array,
2                           size_t      array_size,
3                           const char* separator,
4                           type_t      type)
5  {
6      char* specifier[] = { "%d%s","%f%s", "%lf%s", "%c%s" };
7
8      for (size_t ii = 0; ii < array_size; ++ii) {
9          switch (type) {
10         case int_enumerate:
11         {
12             int* var = array + (ii * sizeof(int));
13             printf(specifier[type], *var, separator);
14         }
15         break;
16         case float_enumerate:
17         {
18             float* var = array + (ii * sizeof(float));
19             printf(specifier[type], *var, separator);
20         }
21         break;
22         case double_enumerate:
23         {
24             double* var = array + (ii * sizeof(double));
25             printf(specifier[type], *var, separator);
26         }
27         break;
28         case char_enumerate:
29         {
30             char* var = array + (ii * sizeof(char));
31             printf(specifier[type], *var, separator);
32         }
33         break;
34         }
35     }
36 }
```

Program 154: Print arrays of several types

Here I need to know the type, not only the size, for a reason: the specifier to print needs the eaxt type. What I do is to declare a `enum` and to be able to index the types in an specifier array. Also, in the `switch` I use a little trick: every case is in a different code block (remember you can create code blocks without them being associated to any control structure), hence in all of them the varuable `var` can be of a different type. This is a step forward, but continues being a very bad function. It is like we had stuck together all the function for every type and we had put them in the same function. It is a step forward, yes, because, even when we have all the same code pieces, they're at least all together.

Ideally we'd want to receive from outside the way we need to print, it could be desirable that the user of the function gave us another one that contained the printing behaviour. That is, in some way we need to turn a behaviour, an algorithm, a function, at the end of the day, in something you can pass, move, translate from one place to another. C has a mechanism to make this that allows us a lot of versatility: function pointers.

A function pointer is a pointer (that is, a memory address) whtat points to instruction that will be executed in said function. These pointers allow us to, as I said before, transfer, communicate, specific behaviours to another parts of our program. Every type of cuntion that can be declared is a different type of pointer. A function, as we saw before, is defined by its return type and the types of the arguments is receives. This means that, for example, these two functions are equal in terms in those terms.

```
1  int sum(int a, int b);
2  int multiply(int x, int y);
```

If every function is a type, you may be thinking that it must have a name to reference is and a way to declare and use it. Yes, but no. Functions cannot be initialized like variables, is you wanted to save the pointer to a function, you could assign is to a `void` pointer. Nevetherless; it is received as argument. To make the function receive a function pointer, this syntax is used.

```
1  tipo_de_retorno (nombre) (tipo1, tipo2...)
2  // for example
3  int(foo)(int, int)
```

Since everything is clearer with an example, let's see something easy: how to make a function that receives another and exeuted it. For example, let's make a function that receives one with this signature: `void (void)` and executes it ten times.

```
1  void execute_10_times(void (foo)()) {
2      for (int ii = 0; ii < 10; ++ii) {
3          foo();
4      }
5  }
```

Program 155: Ejemplo primero de puntero a función como argumento

As you can see, the only diffent thing is the declaration of the function, which we have trated before. The call to `foo` is like any other. We have the other side of the coin, how to call the function `execute_10_times`. This is very simple, because the pointer to a function is simply its name without parenthesis, so the call would be like follows:

```
1  execute_10_times(print_a);
```

Program 156: Call to a function that receives a function pointer

Once this is done, let's come back to the generic printing function, but this time we will make that it receives a function that prints just a single element. We hace to choose the signature of this function, since it is a function that prints, it is only logical that it does not return anything and that receives just the element that we want to print. If it received the element itself, we would have the same problem, we would have to define its type. What we would do is a function that receives a pointer to `void` and return nothing. Beware, this function will be written by the person that uses our generic array printing function, not by us, unless we're the same person, of course. For the example, I will show both functions.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void print_int(void* num)
5  {
6      printf("%d", *((int*)num));
7  }
8
9
10 void print_array_generic(void*       array,
11                          size_t      array_size,
12                          size_t      type_size,
13                          const char* separator,
14                          const char* end,
15                          void(print_foo)(void*))
16 {
17     for (size_t ii = 0; ii < array_size; ++ii) {
18         void* element = array + (ii * type_size);
19         print_foo(element);
20         if (ii != array_size - 1) {
21             printf("%s", separator);
22         }
23     }
24     printf("%s", end);
25 }
26
27 int main(int argc, char** argv)
28 {
29     int array[] = {1,2,3,4,5,6,7,8,9,0};
30     print_array_generic(array,
31                         ARRAY_SIZE(array),
32                         sizeof(*array),
33                         " ",
34                         "\n",
35                         print_int);
36
37 }
```

Program 157: Definición de función de impresión genérica

The generic printing function is very simple, it is simply a loop that iterates over the array and passes the pointer to the printing function that receives as argument. Logically, I need the pointer to the array, the length of it and, since it is a `void` pointer, I need the size of the data type. As you can see, to spice it a bit, I have added two functionalities to the function: a separator character, that will be printed after all the elements (but the last one), and an ending character, that will print after the array. What we could call the specific priting function, that is: `print_int`, it a very simple function, it just calls `printf` performing casting from the pointer and dereferencing it.

It is true that this function model takes us into the same problem, if we want to print different types, we had to define new different functions. This is true, but think about this: we have reduced the amount of duplicated code to its minimum expression, because they're trivial functions of one line. Also, once we are out of basic types, that would be more or less ten specifiers, the code it not duplicated anymore, because any other struct or type would need a custom function written by the creator of such new type.

As you may have noticed, the syntax to declare a function that receives another as argument is complicated and, also, breaks the pattern of an argument list that, up until now, was always a succession of types and names separated by commas. With this syntax for pointers, several parenthesis are included. C allows to define a type for the function pointers. That is, we are discovering yet again another use of `typedef`. Let's see how it would be done and I will commen of some example of functions that would belong to that type.

```
1 typedef void(print_fun_t)(void*); //ej: void print_int(void* a);
2 typedef void(*malloc_t)(void); //ej: void* malloc(void);
3 typedef int(sum_t)(int, int); // ej: int sum(int a, int b);
```
Program 158: Function pointer type definition

Pay attention because, if the function returns a pointer, the asterisk goes inside the parenthesis, next to the type name, not outside. If we made in the program the first definition, we could change our generic function to:

```
1 void print_array_generic(void*      array,
2                          size_t      array_size,
3                          size_t      type_size,
4                          const char* separator,
5                          const char* end,
6                          print_fun_t print_foo);
```
Program 159: Ejemplo final de función que recibe un puntero

It is must clearer, because the last argument is identified, as any other, by a type and a name.

The utlity of this can be very well seen in a kind of functions we have seen before, the sorting functions. Right now those functions sort integer vectors and, also, always from smallest to biggest. This present several improvements, the first one is evident, we have to receive `void` pointers and use them, but the other is more interesting. This second one is: we can use only a way to sort. That is, we can only sort numbers and from smallest to biggest, we can't order structs, we can't comparate strings alphabetically, but we could if we used those new tools. To generalize a sorting function we would need the size of the data type and a function to compare the elements.

Comparison functions are a very concrete type, they're called predicates, and they're functions that return a logic value over a set of arguments. A precidate evaluates a proposition (remember the section about logic), but over the arguments. In this case, a precidate to sort would be: function that checks if the first element is less than the second one. Hence the signature of this funtion would return a logic value and will receive two `void` pointers. Again, it receives two pointer to `void` to be compatible with out generic function, internally, the comparison functions knows which type is behind the `void` pointer.

We are going to use the simpler algorithm we have, the bubble sort, to illustrate this. I am doind this because a generic implementation of, for example, Quick Sort, would be more complex and longer, and I am more interested in the concept of function pointer and `void` pointer. Also, we're going to see an application, sorting of text chain alphabetically.

```c
1  typedef int(comparator_t)(const void*, const void*);
2
3  void generic_swap(void* one, void* other, size_t type_size)
4  {
5      char aux[type_size];
6      memcpy(aux, one, type_size);
7      memcpy(one, other, type_size);
8      memcpy(other, aux, type_size);
9
10 }
11
12 void generic_bubble_sort(void*        array,
13                          size_t       array_size,
14                          size_t       type_size,
15                          comparator_t comparator)
16 {
17     for (int ii = 0; ii < array_size - 1; ++ii) {
18         for (int jj = 0; jj < array_size - 1; ++jj) {
19             void* element = array + (jj * type_size);
20             void* next_element = array + ((jj + 1) * type_size);
21             if (!comparator(element, next_element)) {
22                 generic_swap(element, next_element, type_size);
23             }
24         }
25     }
26 }
```

Program 160: Generic `bubble_sort` definition

As you can see, we define the type of our comparison function, which will return and integer and will receive two constant pointers to `void`, and this is important, the definition of a pointer type hasn't got any implicit conversions. That is: since we have defined the function in such a way it will receive two constant pointers, a function that receives no constant pointers will not be the same type and hence couldn't be used. Mind this when using function pointers.

After that, we have the swap function in its generic function, that is, instead of using the assignment operator, we will use the memory copy operation with the size of the type. Later, the generic sorting function. As you can see, we have simply substituted the old conditional by one that calls the function we received as argument. Remember how bubble sort worked: when the ith element is **greater than** the next one, they swap. That is: when **is it not** true the predicate that the ith element is less than the next one. The advantage that comes from it being a predicate is that we could check if the element is greater, to sort the vector backwards, or use custom predicates for several structs.

Dentro del bucle debemos calcular primero los punteros de los elementos. Esto es así por legibilidad, pero podríamos escribir las expresiones en la propia función de intercambio. Ten en cuenta que debemos multiplicar, de nuevo, `ii` por el tamaño del dato. Recuerda: son punteros a `void`, no entra en juego la aritmética de punteros, son direcciones de memoria absolutas. Una vez calculados simplemente llamamos a la función de intercambio.

Debemos tener en cuenta también la función de comparación. En el caso de un *string* es interesante porque uno puede confundirse debido a que los punteros se suman sobre punteros. Veamos cómo es la función de comparación:

```c
int compare_strings(const void* one, const void* two) {
    char* const* str1 = one, * const* str2 = two;
    return strcmp(*str1, *str2) < 0;
}
```

Program 161: Función auxiliar de comparación de *strings*

Es muy interesante porque puedes ver la primera línea, que introduce algo que no habíamos visto. Esta función recibe dos punteros constantes de `void`. Estos punteros son, en realidad, punteros a punteros a `char`, es decir: `char**`. Pero como los hemos recibido como constantes, no podemos hacerles cásting a ese tipo, el compilador nos diría, hablando claro: «estás haciendo cásting de un puntero constante a uno que no lo es, podrías modificar el contenido». Pero si pusiéramos el modificador `const` primero de todo como hemos hecho siempre el compilador seguiría lanzándonos esa advertencia. La clave es que lo que es constante es lo que, por ejemplo, `one` apunte, es decir, la constancia está pegada al contenido de `one` y `two`. Si escribiéramos `const char**` seguiríamos pudiendo modificar el contenido al que apunta tal dirección. Vamos a verlo con un dibujo:
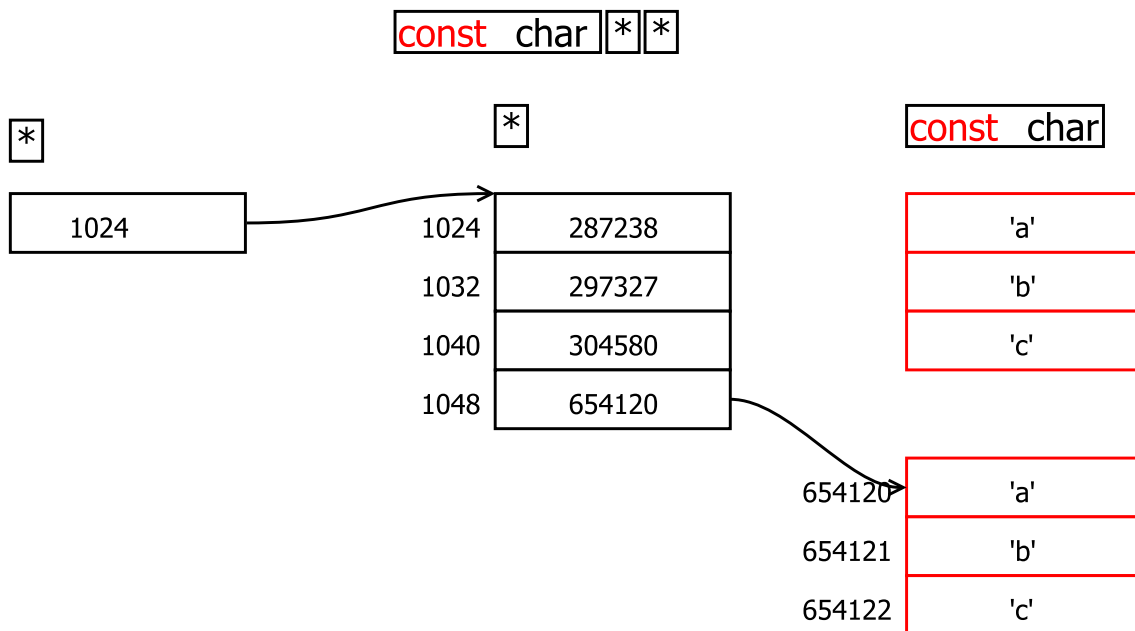

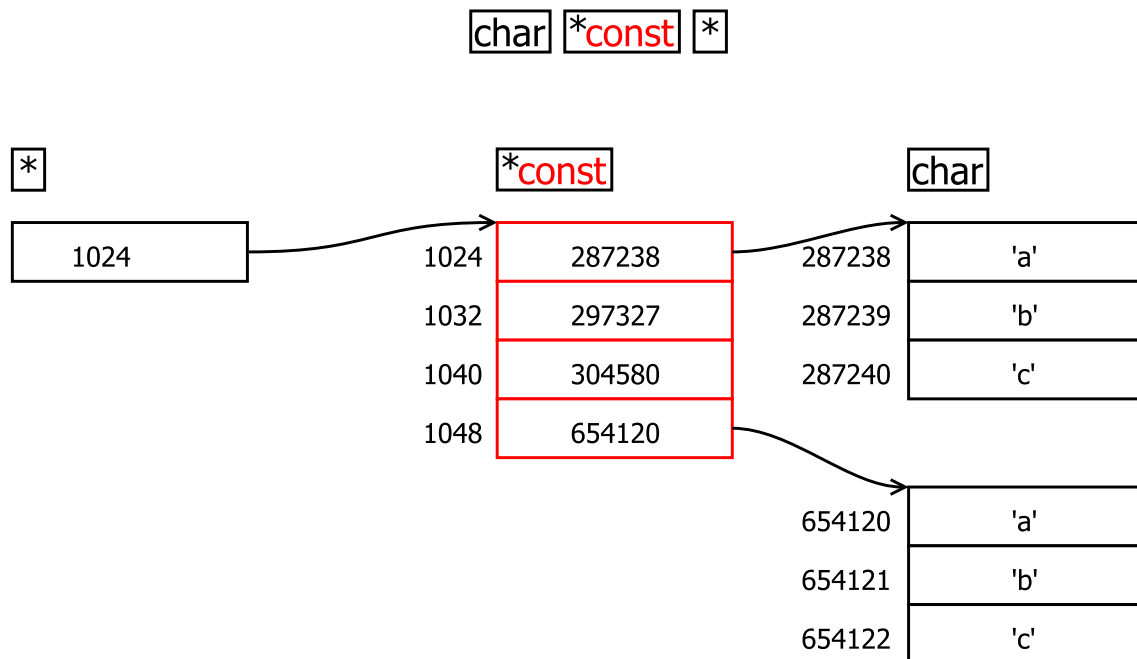Figure 11: Puntero a carácter constante

char *const *

Figure 12: Puntero a puntero constante a `char`

Vamos a explicarlo despacio, si miras el primer dibujo, es «a lo que estamos acostumbrados», cada asterisco es un nuevo nivel de puntero, así que puedes leer la declaración desde la izquierda y construirá los tipos. Empecemos: nos encontramos `const`, lo que venga ahora es constante, después `char`, ahora llega un asterisco, el asterisco inicia un nuevo nivel, así que este puntero **no** será constante, porque no tiene un `const` a la derecha y, finalmente, otro nivel de puntero, que tampoco será constante. Ahora que ya tienes los tres grupos, los inviertes, es decir: puntero no constante, a puntero no contante, a `char` constante.

En el caso siguiente tenemos un `char`, después un asterisco, es decir, un nivel de puntero, lleva const a la derecha, así que es constante, y después otro puntero, sin constancia. Es decir, invirtiéndolo: puntero no constante a puntero constante a char no constante. En ambos casos, en el diagrama, he señalado en rojo los valores que no puedes cambiar, como puedes ver, en el superior no podemos cambiar los *strings*, pero sí los punteros intermedios. En el caso de abajo, por el contrario, podemos modificar los caracteres, pero no los punteros del array intermedio.

Una de las implicaciones de las funciones genéricas es la siguiente: se introduce una sobrecarga inevitable, por dos motivos. El primero es que las funciones que utilizan punteros a `void` tienen que hacer cálculos explícitos que se harían implícitamente. No voy a entrar en detalles de arquitectura de computadores, pero los ordenadores tienen en sus procesadores instrucciones que manejan datos como enteros de cuatro bytes y números decimales (y algunos más). Al tener que copiar byte a byte, impedimos que se utilicen y, además, tenemos que darle más vueltas al bucle de copia, lo cual es más costoso. El segundo es que cuando se llama a una función de manera normal el compilador cuenta con ello para saber cómo generar el binario. Cuando ésta es un argumento, esta tarea se le hace más complicada, porque no sabe qué función es hasta el momento de la ejecución. Para hacer esto patente, vamos a hacer una comparación con el tiempo que tarden ambas versiones en ordenar 65.536 y 131.072 elementos. Vamos a comparar ambas cargas de trabajo porque quiero que veas una cosa.

Función	N=35.536	N=131.072
Específica	16,21	64,61
Genérica	41,20	164,44
Ratio	0,39	0,39

Table 12: Tiempos de ejecución de los distintos algoritmos

Como puedes ver, la versión genérica tarda más, pero he calculado un dato importante a ese respecto: el ratio entre el tiempo del algoritmo específico y el algoritmo genérico. Como puedes ver, aunque el algoritmo genérico es peor que el específico, la buena noticia es que esa diferencia es constante, es decir: no empeora con el tamaño del vector. Esto hace que, si podemos asumir el aumento de tiempo, la solución sea escalable, que es una manera que se tiene en informática de decir que puedes hacer crecer algo sin quedarte sin recursos rápidamente.

Un ejercicio muy interesante sería que programaras la versión genérica de *Quick Sort* y que, además, hicieras estas mismas mediciones. Para medir el tiempo puedes utilizar este código:

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

double timespec_to_double(const struct timespec* tm)
{
    return tm->tv_sec + tm->tv_nsec / 1000000000.0;
}

int main(int argc, char** argv)
{
    double start, stop;
    struct timespec start_ts, stop_ts;

    clock_gettime(CLOCK_REALTIME, &start_ts);
    start = timespec_to_double(&start_ts);
    // Aquí el código que quieres medir.
    clock_gettime(CLOCK_REALTIME, &stop_ts);
    stop = timespec_to_double(&stop_ts);
    printf("Hemos tardado: %lf\n", stop - start);
}
```

Program 162: Cómo medir el tiempo

La función `clock_gettime` es una función para medir el tiempo de un modo peculiar, en sistemas Linux se mide el tiempo desde el primero de enero de 1970. Así, la estructura `timespec` indica el tiempo pasado desde entonces como un conjunto de segundos más los correspondientes nanosegundos en sus dos miembros. Como eso es poco práctico he creado una pequeña función para convertirlo a número decimal y así poder restarlo cómodamente. Después, simplemente mido el tiempo antes y después del código que quiero saber cuánto tarda y los resto, como puedes ver.

17.   Ejemplo completo de programa

Esta sección está al final porque, si hasta ahora hemos visto cada parte del lenguaje en detalle y por sí misma, en esta vamos a intentar montar todas las piezas en una gran fotografía. Para esto vamos a utilizar y refinar un ejemplo que ha sido recurrente en el manual: la gestión de una estructura que almacena los datos de una persona, pero vamos a conseguir separar bien al usuario de la funcionalidad interna del código que se encague de eso.

Lo que haremos es crear un archivo de código fuente llamado `person.c` y su correspondiente archivo de cabeceras, `person.h`, en este archivo incluiremos funcionalidad para crear una estructura persona, cambiar sus atributos, leerlos y serializarla. Además, vamos a ver un interesante artefacto del lenguaje para poder impedir que el usuario se entrometa en nuestra estructura y pueda alterar los datos de manera incorrecta. Por ejemplo: asignando los punteros a una zona de memoria que no controlemos desde estas funciones proporcionadas para manipular la estructura de datos.

Lo primero que voy a hacer es crear el archivo de cabeceras porque ya hemos definido de una manera muy concreta la funcionalidad de este código fuente. Aquí hay una cosa interesante que podremos comentar, veamos el archivo:

```
1  #ifndef PERSON_H
2  #define PERSON_H
3
4  typedef struct person_s person_t;
5
6  person_t *create_person(const char *name, const char *last_name,
7                          unsigned int age);
8
9  void destroy_person(person_t *p);
10
11 void person_set_name(person_t *p, const char *name);
12
13 void person_set_last_name(person_t *p, const char *last_name);
14
15 void person_set_age(person_t *p, unsigned int age);
16
17 const char *person_get_name(const person_t *person);
18
19 const char *person_get_last_name(const person_t *person);
20
21 unsigned int person_get_age(const person_t *person);
22
23 char *person_to_string(const person_t *p);
24
25 #endif
```
Program 163: Ejemplo final de programa – `person.h`

Y aquí puedes ver una de las cosas interesantes de este ejemplo final: estamos declarando el tipo `person_t`, pero no el *struct* al que da nombre, esto quiere decir que cualquier archivo de código fuente que incluya este **no** podrá saber la definición de tal *struct*. La implicación de esto es que no podrá declarar variables de este tipo, tan solo punteros, puede declarar un puntero, porque todos los punteros tienen el mismo tamaño. Si intentáramos declarar una variable de este tipo, el compilador lanzaría un error como el siguiente:

```
main.c: In function 'main':
main.c:5:14: error: storage size of 'francis' isn't known
    5 |     person_t francis;
      |              ^~~~~~~
```

Este es el mecanismo que nos permite impedir que el usuario altere el contenido de la estructura fuera de nuestro control (como comentamos en el programa 64) porque, del mismo modo que no conoce el tamaño del tipo, tampoco conoce los miembros de esta estructura, así que no puede accederse a ellos. Nota, además, como no hemos incluido ninguna cabecera en `person.h`. Si necesitáramos cabeceras, por ejemplo, la cabecera `stdint.h` contiene definiciones de tipo útiles como aquéllos de tamaño fijo: `int8_t`, `int16_t`, etc.; si quisiéramos definir alguna función con un argumento de este tipo o de tipo de retorno, sí sería necesario que incluyéramos esta cabecera. Si las necesitamos en las implementaciones (en las declaraciones de esctructuras, en las definiciones de funciones...), será en el archivo de código fuente (en el `.c`) donde las incluiremos.

El siguiente archivo es, precisamente, este archivo de código fuente: `person.c`. Es bastante largo, así que vamos a incluirlo en tres secciones: la sección de declaración de tipos (que sólo contendrá uno), las funciones de manipulación del contenido de la estructura y, finalmente, la de recuperación de la información.

```c
1  #include <string.h> //strdup, memset
2  #include <stdlib.h> //malloc
3  #include <stdio.h>  //snprintf
4  #include "person.h"
5
6  struct person_s
7  {
8      char *name;
9      char *last_name;
10     unsigned int age;
11 };
12
13 person_t *create_person(const char *name,
14                         const char *last_name,
15                         unsigned int age)
16 {
17     person_t *res = malloc(sizeof(*res));
18     memset(res, 0, sizeof(*res));
19
20     res->age = age;
21     res->name = strdup(name);
22     res->last_name = strdup(last_name);
23
24     return res;
25 }
26
27 void destroy_person(person_t *p)
28 {
29     free(p->name);
30     free(p->last_name);
31     free(p);
32 }
```

Program 164: Ejemplo final de programa – `person.c` definiciones

Aquí podemos ver la defininión del tipo del que en la cabecera hicimos un `typedef`, este estilo de declaración de un tipo se llama declaración anticipada o, en inglés, *forwarding declaration*. Aquí, aparte de la definición del tipo propiamente dicho, tenemos las funciones que lo crean y que lo destruyen. Como esta estructura contiene elementos reservados con memoria dinámica, debemos proveer al usuario una manera de liberar los recursos de la estructura. Como puedes ver, en las funciones de creación reservamos espacio **para la propia estructura** y para sus campos.

Debemos reservar nosotros dinámicamente la estructura aparte de sus campos porque, recordemos, fuera de este archivo de código fuente no podremos declarar más que punteros, y ese puntero no tendrá espacio para nada si no lo declaramos. Después, reservamos memoria para el contenido al que apuntarán los **miembros** de la estructura. En la función de destrucción, simétricamente, liberamos primero los contenidos y después la propia estructura. Nota, además, cómo hemos declarado todos los argumentos que hemos podido como constantes, para que el usuario no tenga dudas de si vamos a modificar datos que nos proporcione.

Las siguientes funciones son las que nos permiten sobreescribir los datos:

```c
1  void person_set_name(person_t *p, const char *name)
2  {
3      free(p->name);
4      p->name = strdup(name);
5  }
6
7  void person_set_last_name(person_t *p, const char *last_name)
8  {
9      free(p->last_name);
10     p->last_name = strdup(last_name);
11 }
12
13 void person_set_age(person_t *p, unsigned int age)
14 {
15     p->age = age;
16 }
```

Program 165: Ejemplo final de programa – `person.c` manipulación

Como puedes ver, las funciones son simples, liberamos la memoria de los campos y después le asignamos la duplicación del argumento que se nos pasa. De nuevo, observa cómo hemos definido como constantes los argumentos del mismo modo que hicimos en la función de creación. Las funciones no devuelven nada (`void`) porque no tendríai sentido. Aunque siempre podrían devolver un entero que actuara como código de error, por ejemplo si la reserva de memoria fallara, se podría indicar devolviendo un número menor que cero.

```
1  const char *person_get_name(const person_t *p)
2  {
3      return p->name;
4  }
5
6  const char *person_get_last_name(const person_t *person)
7  {
8      return person->last_name;
9  }
10
11 unsigned int person_get_age(const person_t *person)
12 {
13     return person->age;
14 }
15
16 char *person_to_string(const person_t *p)
17 {
18 #define MAX_STRING_SIZE ((unsigned int)1024)
19
20     char res[MAX_STRING_SIZE + 1];
21     snprintf(res, MAX_STRING_SIZE, "{\"name\":\"%s\","
22                                     "\"last_name\":\"%s\","
23                                     "\"age\":%u}",
24             p->name, p->last_name, p->age);
25     return strdup(res);
26 #undef MAX_STRING_SIZE
27 }
```

Program 166: Ejemplo final de programa – `person.c` recuperación

Aquí debes notar que devolvemos punteros contantes a `char`, precisamente para impedir que el usuario libere, manipule o cambie el contenido de los campos del *struct*. Sin embargo; en la función de serialización (que he reducido a su versión más simple) devuelvo un puntero no constante porque la responsabilidad de liberar es del usuario de la funcionalidad, no de esta biblioteca. Además, en esta última función puedes ver que podemos **eliminar** una macro con la directiva `#undef`. Esto es útil cuando necesitas inicializar un array, como aquí, pero no quieres contaminar de símbolos el código fuente. Así, si otra función usara strings de otro tamaño, podríamos usar el mismo nombre, como si la macro fuera una variable distinta. De nuevo: ten cuidado, las macros trabajan a nivel de preprocesado, por lo que no estás definiendo ninguna variable en la función, sólo una región de código donde un símbolo se sustituirá por otro.

Finalmente, en el archivo principal podemos utilizar la funcionalidad:

```
1  #include "person.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(void)
6  {
7      person_t* person = create_person("John", "Smith", 18);
8
9      char* serialization = person_to_string(person);
10
11     printf("%s\n", serialization);
12     free(serialization);
13
14     person_set_name(person, "Michael");
15     person_set_last_name(person, "Johnson");
16     person_set_age(person, 33);
17
18     serialization = person_to_string(person);
19     printf("%s\n", serialization);
20     free(serialization);
21 }
```

Program 167: Ejemplo final de programa – `main.c`

Aquí se puede ver cómo se utilizan estructuras con este patrón de diseño. Primero la reservas, después la usas, la puedes manipular y, finalmente, la liberas, todo ello con las funciones proporcionadas junto con el tipo de dato. Con este patrón, el usuario de la funcionalidad que hemos programado tiene menos capacidad para «hacer algo mal».

Ahora, vamos a ver rápidamente cómo se podría compilar, para recordarlo. Primero lo haremos utilizando el código objeto y, después, crearemos una biblioteca dinámica y la enlazaremos. Para compilar utilizando el código objeto seguiremos estos pasos:

1. Crear el código objeto de `person.c`

   ```
   $ gcc -c person.c
   ```

2. Crear el código objeto de `main.c`

   ```
   $ gcc -c main.c
   ```

3. Crear el ejecutable con ambos códigos objeto

   ```
   $ gcc -o main.exe main.o person.o -g -Wall -Wextra
   ```

Para la biblioteca, seguiremos estos pasos:

1. Crear el código objeto de `person.c`

```
$ gcc -c person.c
```

2. Crear una biblioteca con este código objeto:

```
$ gcc -shared -o libperson.so person.o
```

3. Crear el código objeto de `main.c`

```
$ gcc -c main.c
```

4. Crear el ejecutable usando la biblioteca:

```
$ gcc -L. -Wl,-rpath=. -o main.exe main.o -lperson
```

En este ejemplo final se han visto ejemplos de la mayoría de conceptos que se han explicado en el manual: variables, punteros, memoria, reserva dinámica, estructuras, macros, enlazado, compilación y constancia y signo. Es mucha información en pocas páginas, pero permite tener una foto global de todo si ya se ha leído antes con detenimiento.

18.   Anexo A: soluciones a ejercicios

Ex. 1: Escribe un programa y declara en él una estructura que defina un círculo en dos dimensiones (su centro y su radio). Y haz que el programa declare una variable de ese tipo y calcule su área.

```c
#include <stdio.h>

struct circle_s {
    double x;
    double y;
    double r;
};

int main(void)
{
    struct circle_s circle = { 1 , 1 , 3.4 };
    double area = 3.141592 * circle.r * circle.r;
    printf("El área del círculo en el punto [%f, %f] con un radio de %
        f es: %f\n", circle.x, circle.y, circle.r, area);
}
```

Program 168: Solución al ejercicio 1

Ex. 2: Haz un programa que, basándose en el struct punto presentado en el ejemplo, declare e inicialice un array de ellos y vaya diciendo las direcciones que hay que seguir para ir de uno a otro.

```c
1  #include <stdio.h>
2  struct point_s {
3      double x;
4      double y;
5  };
6
7  int main(void)
8  {
9      struct point_s points[10] = { {-1.056171, 3.401877},
10                                    {2.984400, 2.830992},
11                                    {-3.024486, 4.116474},
12                                    {2.682296, -1.647772},
13                                    {0.539700, -2.222253},
14                                    {1.288709, -0.226029},
15                                    {0.134009, -1.352155},
16                                    {4.161951, 4.522297},
17                                    {2.172969, 1.357117},
18                                    {1.069689, -3.583974} };
19
20      for(int ii = 1; ii < 10; ++ii){
21          if (points[ii - 1].x < points[ii].x) {
22              printf("Derecha");
23          }else if(points[ii - 1].x == points[ii].x){
24              printf("Quieto");
25          }else if(points[ii - 1].x > points[ii].x){
26              printf("Izquierda");
27          }
28          printf(", ");
29          if (points[ii - 1].y < points[ii].y) {
30              printf("Arriba");
31          }else if(points[ii - 1].y == points[ii].y){
32              printf("Quieto");
33          }else if(points[ii - 1].y > points[ii].y){
34              printf("Abajo");
35          }
36          printf("\n");
37      }
38  }
```

Program 169: Solución al ejercicio 2

Ex. 3: Haz un programa que declare un array bidimensional y calcule la suma de sus filas y sus columnas.

```c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int array[3][3] = { {1,3,6},{7,3,6},{1,2,4} };
6
7      for (int ii = 0; ii < 3; ++ii) {
8          for (int jj = 0; jj < 3; ++jj) {
9              printf("%d ", array[ii][jj]);
10         }
11         int suma = 0;
12         for(int jj = 0; jj < 3; ++jj){
13             suma += array[ii][jj];
14         }
15         printf("= %d\n", suma);
16     }
17     for(int ii = 0; ii < 3*2; ++ii){
18         printf("-");
19     }
20     printf("\n");
21     for(int ii = 0; ii < 3; ++ii){
22         int suma = 0;
23         for(int jj = 0; jj < 3; ++jj){
24             suma+=array[jj][ii];
25         }
26         printf("%d ", suma);
27     }
28     printf("\n");
29 }
```

Program 170: Solución al ejercicio 3

Ex. 4: Haz un programa que haga lo siguiente para los números del 1 al 100 ambos incluidos: si el número es divisible entre 2, debe imprimirse por pantalla «fizz», si es divisible entre 5, «buzz», y si es divisible entre los dos, «fizzbuzz», no imprimir nada en otro caso.

```c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      for (int ii = 1; ii <= 100; ++ii){
6          int end_of_line = 0;
7          if (ii % 2 == 0){
8              printf("fizz");
9              end_of_line = 1;
10         }
11
12         if(ii % 5 == 0){
13             printf("buzz");
14             end_of_line = 1;
15         }
16         if(end_of_line){
17             printf("\n");
18         }
19     }
20 }
```

Program 171: Solución al ejercicio 4

Ex. 5: Escribe una función que calcule si un número es primo o no.

```c
1  #include <stdio.h>
2
3  int is_prime(int number) {
4      int prime = 1;
5      for (int ii = 2; ii < number / 2 && prime; ++ii) {
6          if (0 == number % ii) {
7              prime = 0;
8          }
9      }
10     return prime;
11 }
12
13 int main(void)
14 {
15     for(int ii = 2; ii < 100; ii++){
16         printf("El número %d ", ii);
17         if(is_prime(ii)){
18             printf("es primo.");
19         }else{
20             printf("no es primo");
21         }
22         printf("\n");
23     }
24 }
```

Program 172: Solución al ejercicio 5

Ex. 6: Escribe una función que calcule la distancia entre dos estructuras punto de las usadas en la sección anterior.

```c
#include <stdio.h>
#include <math.h>
struct point_s {
    double x;
    double y;
};

double distance(struct point_s a, struct point_s b) {
    double res = 0.0;
    double diff_x = a.x - b.x;
    double diff_y = a.y - b.y;
    res = sqrt(diff_x * diff_x + diff_y * diff_y);
    return res;
}

int main(void)
{
    struct point_s a = {1.2, 4.3};
    struct point_s b = {3.4, 5.5};
    printf("La distancia entre [%f, %f] y [%f, %f] es: %f\n", a.x, a.y,
        b.x, b.y, distance(a,b));
}
```

Program 173: Solución al ejercicio 6

Ex. 7: Escribe una función que reciba un array de enteros y un caracter separador que imprima los elementos del array separados por ese caracter.

```c
#include <stdio.h>
void print_separated(int array[], int array_size, char separator){
    for(int ii = 0; ii < array_size; ++ii){
        printf("%d%c", array[ii], separator);
    }
}

int main(void)
{
    int my_array[] = {1,2,3,4,5,6,7,8,9,0};
    print_separated(my_array, 10, '|');
    printf("\n");
}
```

Program 174: Solución al ejercicio 7

Ex. 8: Escribe una función que encapsule el programa 17: Program solving a linear equations system with conditionals. La función debe recibir los coeficientes de las ecuaciones (a, b, c, d, e y f). Puede recibirlos por separado o en un array. Para devolver el resultado puedes crear una estructura que simplemente tenga dos `double`.

```c
#include <stdio.h>

struct solution_s {
```

```c
 4       double x;
 5       double y;
 6       int solved;
 7 };
 8
 9 struct solution_s linear_system(int a, int b, int c, int d, int e, int
      f) {
10
11       double divisor;
12       struct solution_s res;
13       res.solved = 1;
14       if (a != 0 && d != 0) {
15           divisor = (a * e - d * b);
16           if (divisor == 0)
17           {
18               printf("El sistema es irresoluble .\n");
19               res.solved = 0;
20           }
21           else
22           {
23               res.y = (a * f - d * c) / divisor;
24               res.x = (f - e * res.y) / (d);
25           }
26       }
27       else if (b != 0 && e != 0) {
28           divisor = (b * d - e * a);
29           if (divisor == 0) {
30               printf("El sistema es irresoluble .\n");
31               res.solved = 0;
32           }
33           else {
34               res.x = (b * f - e * c) / divisor;
35               res.y = (c - a * res.x) / b;
36           }
37       }
38       else if ((a == 0 && b == 0) || (d == 0 && e == 0)) {
39           printf(" Esto no es un sistema \n");
40           res.solved = 0;
41       }
42       else {
43           if (a != 0) {
44               res.x = (double)c / a;
45               res.y = (double)f / e;
46           }
47           else {
48               res.x = (double)f / d;
49               res.y = (double)c / b;
50           }
51       }
52       return res;
53 }
54
55
```

```
56  int main(void)
57  {
58      struct solution_s sol = linear_system(1, 1, 1, 2, 2, 2);
59      printf(" %dx+ %dy= %d\n", 1, 2, 3);
60      printf(" %dx+ %dy= %d\n", 4, 5, 6);
61      if (sol.solved) {
62          printf("x = %f; y = %f\n", sol.x, sol.y);
63      }
64      else {
65          printf("El sistema no tiene solucion.\n");
66      }
67  }
```

Program 175: Solución al ejercicio 8

Ex. 9: Escribe una función que normalice los elementos de un array de `double`.

```
1   #include <stdio.h>
2
3   void normalize(double array[], int array_size) {
4       double biggest = array[0];
5       for (int ii = 1; ii < array_size; ++ii) {
6           if (array[ii] > biggest) {
7               biggest = array[ii];
8           }
9       }
10      for (int ii = 0; ii < array_size; ++ii) {
11          array[ii] /= biggest;
12      }
13  }
14
15  int main(void)
16  {
17      double array[] = { 1,2,3,4,5,6,7,8,9,10 };
18      normalize(array, 10);
19      for(int ii = 0; ii < 10; ++ii){
20          printf("%f\n", array[ii]);
21      }
22      printf("\n");
23  }
```

Program 176: Solución al ejercicio 9

Ex. 10: Completa esta tabla de números en diferentes bases numéricas:

Decimal	Binario	Hexadecimal
73	0100 1001	0x049
38	0010 0110	0x026
303	0001 0010 1111	0x12F
128	1000 0000	0x080

Ex. 11: Vuelve al ejercicio noveno y reproduce los contenidos de la pila en cada bloque de código del programa. Utiliza de referencia la solución que propongo yo.

1. Función main

   1. Array (10 elementos)

   2. Entramos en la función normalize

      1. Array (puntero a)

      2. `array_size`

      3. `biggest`

      4. Primer bucle for

         1. `ii`

      5. Segundo bucle for

         1. `ii`

   3. Bucle for

      1. `ii`

Ex. 12: Haz un programa que cree un puntero de tres niveles de tipo `int`, lo reserve correctamente, lo rellene con el valores correlativos **empezando en uno** y después lo imprima de una manera comprensible. Finalmente, libéralo también de tal modo que no quede memoria sin liberar al final del programa.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define DEPTH (10)
5  #define WIDTH (5)
6  #define HEIGHT (12)
7
8  int main(int argc, char const *argv[]) {
9      int ***cube = malloc(sizeof(*cube) * DEPTH);
10
11     for (int ii = 0; ii < DEPTH; ++ii) {
12         cube[ii] = malloc(sizeof(**cube) * HEIGHT);
13         for (int jj = 0; jj < HEIGHT; ++jj) {
14             cube[ii][jj] = malloc(sizeof(***cube) * WIDTH);
15             for (int kk = 0; kk < WIDTH; ++kk) {
16                 cube[ii][jj][kk] =
17                     kk + jj * WIDTH + ii * HEIGHT * WIDTH + 1;
18             }
19         }
20     }
21
22     for (int ii = 0; ii < DEPTH; ++ii) {
23         for (int jj = 0; jj < HEIGHT; ++jj) {
24             for (int kk = 0; kk < WIDTH; ++kk) {
25                 printf("%3d ", cube[ii][jj][kk]);
26             }
27             printf("\n");
28         }
29         printf("\n");
30     }
31
32     for (int ii = 0; ii < DEPTH; ++ii) {
33         for (int jj = 0; jj < HEIGHT; ++jj) {
34             free(cube[ii][jj]);
35         }
36         free(cube[ii]);
37     }
38     free(cube);
39
40     return 0;
41 }
```

Program 177: Solución al ejercicio 12

Ex. 13: Basándote en el programa anterior, crea dos funciones, una para crear una matriz tridimensional con memoria dinámica dadas sus tres dimensiones y otra para liberarla.

```
1  int ***malloc_cube(size_t depth, size_t height, size_t width) {
2      int ***cube = malloc(sizeof(*cube) * depth);
3
4      for (int ii = 0; ii < depth; ++ii) {
5          cube[ii] = malloc(sizeof(**cube) * height);
6          for (int jj = 0; jj < height; ++jj) {
7              cube[ii][jj] = malloc(sizeof(***cube) * width);
8              for (int kk = 0; kk < width; ++kk) {
9                  cube[ii][jj][kk] =
10                     kk + jj * width + ii * height * width + 1;
11             }
12         }
13     }
14     return cube;
15 }
```

Program 178: Solución al ejercicio 13 – reserva

```
1  void print_cube(int ***cube, size_t depth, size_t height,
2                  size_t width) {
3      for (int ii = 0; ii < depth; ++ii) {
4          for (int jj = 0; jj < height; ++jj) {
5              for (int kk = 0; kk < width; ++kk) {
6                  printf("%3d ", cube[ii][jj][kk]);
7              }
8              printf("\n");
9          }
10         printf("\n");
11     }
12 }
```

Program 179: Solución al ejercicio 13 – impresión

```
1  void free_cube(int ***cube, size_t depth, size_t height,
2                 size_t width) {
3      for (int ii = 0; ii < depth; ++ii) {
4          for (int jj = 0; jj < height; ++jj) {
5              free(cube[ii][jj]);
6          }
7          free(cube[ii]);
8      }
9      free(cube);
10 }
```

Program 180: Solución al ejercicio 13 – liberación

```
1  int main(int argc, char const *argv[]) {
2      int ***cube = malloc_cube(DEPTH, HEIGHT, WIDTH);
3      print_cube(cube, DEPTH, HEIGHT, WIDTH);
4      free_cube(cube, DEPTH, HEIGHT, WIDTH);
5  }
```

Program 181: Solución al ejercicio 13 – función `main`

Ex. 14: Escribe un programa que reciba un número variable de números como argumentos e imprima la descomposición en factores primos de todo ellos. Se recomienda hacer control de errores comprobando que los argumentos son números antes de utilizarlos, etc.

```
1  int main(int argc, char const *argv[]) {
2      int ***cube = malloc_cube(DEPTH, HEIGHT, WIDTH);
3      print_cube(cube, DEPTH, HEIGHT, WIDTH);
4      free_cube(cube, DEPTH, HEIGHT, WIDTH);
5  }
```

Program 182: Solución al ejercicio 13 – función `main`

Ex. 15: Escribe un programa que lea **por consola** una serie de palabras y que sólo deje de leer cuando se introduzca «!!» como palabra. Después, debe imprimir dichas palabras en orden aleatorio. La función `rand` devuelve un número aleatorio entre cero y el máximo entero positivo. Si quieres que devuelva números aleatorios **distintos** cada vez debes ejecutar `srand(time(NULL));` al inicio de la función `main`. Debes incluir la cabecera `time.h`.

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

#define STRING_SIZE (1024)
#define MAX_WORDS (1024)

int main(int argc, char const *argv[]) {
    char *word_set[1024];
    int word_length = 0;
    do {
        word_set[word_length] = malloc(STRING_SIZE);
        scanf("%s", word_set[word_length]);
        word_length++;
    } while (strcmp(word_set[word_length - 1], "!!"));

    srand(time(NULL));
    for (int ii = 0; ii < word_length - 1; ++ii) {
        char *aux           = word_set[ii];
        int rand_index      = rand() % (word_length - 1);
        word_set[ii]        = word_set[rand_index];
        word_set[rand_index] = aux;
    }
    for (int ii = 0; ii < word_length - 1; ++ii) {
        printf("%s\n", word_set[ii]);
        free(word_set[ii]);
    }
    free(word_set[word_length-1]);
}
```

Program 183: Solución al ejercicio 15

Como nota, para «barajar» el vector de palabras lo que hago el recorrerlo intercambiando cada palabra con una posición aleatoria. Hay otros métodos que quizás hayas usado como generar una posición aleatoria del vector y copiarlo a otro, el problema de esto es que si lo que haces es generar un índice nuevo cuando encuentras que ya has copiado ese, el número de veces que ejecutas el aleatorio es, lógicamente, impredecible. Tal y como lo he escrito yo el algoritmo siempre tardará lo mismo generando resultados moderadamente aleatorios.

Ex. 16: Haz una función que lea dos archivos e **intercambie** su contenido, escribe dicho programa de tal modo que no sea necesario alojar ninguno de los dos archivos en memoria completamente.

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(int argc, char const *argv[]) {
5       FILE *file_1 = NULL, *file_2 = NULL, *file_aux = NULL;
6       const int BLOCK_SIZE = 1024;
7       int read              = 0;
8       char buffer[BLOCK_SIZE], aux_file_path[] = "/tmp/auxFile.txt";
9       if (argc < 3) {
10          printf("Uso: main.exe <archivo1> <archivo2>\n");
11          return EXIT_FAILURE;
12      }
13
14      file_1 = fopen(argv[1], "r+");
15      if (NULL == file_1) {
16          printf("ERROR: El primer archivo no existe.\n");
17          return EXIT_FAILURE;
18      }
19
20      file_2 = fopen(argv[2], "r+");
21      if (NULL == file_2) {
22          printf("ERROR: El segundo archivo no existe.\n");
23          fclose(file_1);
24          return EXIT_FAILURE;
25      }
26
27      file_aux = fopen(aux_file_path, "w+");
28      if (NULL == file_aux) {
29          fclose(file_1);
30          fclose(file_2);
31          return EXIT_FAILURE;
32      }
33
34      // copy file 1 to aux
35      while (read = fread(buffer, sizeof(char), BLOCK_SIZE, file_1)) {
36          fwrite(buffer, sizeof(char), read, file_aux);
37      }
38      fclose(file_1);
39      file_1 = fopen(argv[1], "w+");
40      if (NULL == file_1) {
41          printf("Error, el primer archivo no se ha podido reabrir\n");
42      }
43
44      // copy file 2 to 1
45      while (read = fread(buffer, sizeof(char), BLOCK_SIZE, file_2)) {
46          fwrite(buffer, sizeof(char), read, file_1);
47      }
48
49      fclose(file_2);
50      file_2 = fopen(argv[2], "w+");
51      if (NULL == file_2) {
52          printf("Error, el segundo archivo no se ha podido reabrir\n");
53      }
```

```
54
55      // copy aux file to file 2, we need to go back to begin of file aux
56      fseek(file_aux, 0, SEEK_SET);
57      while (read = fread(buffer, sizeof(char), BLOCK_SIZE, file_aux)) {
58          fwrite(buffer, sizeof(char), read, file_2);
59      }
60
61      fclose(file_1);
62      fclose(file_2);
63      fclose(file_aux);
64      remove(aux_file_path);
65  }
```

Program 184: Solución al ejercicio 16

Ex. 17: Escribe una función que reciba una palabra como argumento e indique en qué posición (en bytes) dentro del archivo se encuentra la palabra. Sólo tienes que dar la primera ocurrencia, si la palabra no se encuentra, devuelve un número negativo. Haz un programa que, con esa función, reciba una ruta a un archivo y una palabra e imprima el resultado de buscar la palabra en el archivo.

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

int find_in_file(const char *path, const char *word) {
    FILE *file = NULL;
    char *buffer;
    int word_length = 0, read = 0, pos = -1;

    file = fopen(path, "r+");
    if (NULL == file) {
        printf("ERROR: El archivo no existe.\n");
        return -1;
    }
    word_length = strlen(word);
    buffer      = malloc(sizeof(char) * word_length * 2);

    while (read = fread(buffer, sizeof(char), word_length * 2, file)) {
        fseek(file, word_length - read, SEEK_CUR);
        for (int ii = 0; ii < word_length; ++ii) {
            char local_word[word_length + 1];
            memcpy(local_word, buffer + ii, word_length);
            local_word[word_length] = 0;
            if (!strcmp(local_word, word)) {
                pos = ftell(file) + ii - word_length;
                goto end;
            }
        }
    }
end:
    free(buffer);
    fclose(file);
    return pos;
}

int main(int argc, char const *argv[]) {

    int pos = find_in_file(argv[1], argv[2]);
    printf("La palabra %s está en la posición %d en el archivo %s\n",
            argv[2], pos, argv[1]);
}
```

Program 185: Solución al ejercicio 17

Aquí puedes ver un uso típico de la instrucción `goto`, como necesitamos hacer lo mismo encontremos la palabra o no, lo que hacemos es establecer una etiqueta y saltar allí para liberar recursos y devolver el resultado.

Ex. 18: Reescribe el ejercicio 15 prescindiendo del array estático de punteros a `char`. (Usa `realloc` y `strdup`.

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

int main(int argc, char const *argv[]) {
    char word[1024];
    char **word_set = NULL;
    int word_length = 0;
    do {
        word_set = realloc(word_set, ++word_length * sizeof(char *));
        scanf("%s", word);
        word_set[word_length - 1] = strdup(word);
    } while (strcmp(word, "!!"));

    srand(time(NULL));
    for (int ii = 0; ii < word_length - 1; ++ii) {
        char *aux           = word_set[ii];
        int rand_index      = rand() % (word_length - 1);
        word_set[ii]        = word_set[rand_index];
        word_set[rand_index] = aux;
    }
    for (int ii = 0; ii < word_length - 1; ++ii) {
        printf("%s\n", word_set[ii]);
    }
    for (int ii = 0; ii < word_length; ++ii) {
        free(word_set[ii]);
    }
    free(word_set);
}
```

Program 186: Solución al ejercicio 18

Ex. 19: Escribe un programa que reciba un número indeterminado de palabras como argumentos y los ordene alfabéticamente y que, después, los imprima.

```c
1  void generic_swap(void *one, void *other, size_t type_size) {
2  //...
3
4  void generic_bubble_sort(void *array, size_t array_size,
5  //...
6
7  int compare_strings(const void *one, const void *two) {
8  //...
9
10 int compare_strings(const void* one, const void* two) {
11 //...
12
13 int main(int argc, char const *argv[]) {
14
15     generic_bubble_sort(argv + 1, argc - 1, sizeof(char *),
16                         compare_strings);
17
18     for (int ii = 1; ii < argc; ++ii) {
19         printf("%s\n", argv[ii]);
20     }
21 }
```

Program 187: Solución al ejercicio 19

He usado las funciones de ejemplo para ordenar, así que omito su contenido, simplemente tenemos que utilizar el comparador adecuado y tener en cuenta que el primer argumento es el nombre de programa, que no queremos ordenar. Además, puedes ver que podemos modificar el orden de los argumentos, pero no su contenido, al haber declarado `argv` como `char const*argv[]` que quiere decir un array (puntero) no constante a `char` constante. Es decir, como ya vimos en la figura 11.

Ex. 20: Haz un programa que reciba como argumento una palabra y un número. Si el número es cero, debe convertir la palabra a minúscula, si el número es distinto de cero, debe convertirla a mayúscula.

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   char char_to_upper_case(char c) {
5       if (c < 123 && c > 96) {
6           return c - 32;
7       }
8       return c;
9   }
10
11  char char_to_lower_case(char c) {
12      if (c < 91 && c > 64) {
13          return c + 32;
14      }
15      return c;
16  }
17
18  char string_to_upper_case(char *message) {
19      int length = strlen(message);
20      for (int ii = 0; ii < length; ++ii) {
21          message[ii] = char_to_upper_case(message[ii]);
22      }
23  }
24
25  char string_to_lower_case(char *message) {
26      int length = strlen(message);
27      for (int ii = 0; ii < length; ++ii) {
28          message[ii] = char_to_lower_case(message[ii]);
29      }
30  }
31
32  int main(int argc, char const *argv[]) {
33
34      char *message = strdup(argv[1]);
35      int code = atoi(argv[2]);
36      if(code){
37          string_to_upper_case(message);
38      }else{
39          string_to_lower_case(message);
40      }
41      printf("%s\n", message);
42      free(message);
43  }
```

Program 188: Solución al ejercicio 20

Aquí hemos utilizado dos funciones diferentes para poner a mayúscula y minúscula, otra opción sería utilizar un parámetro lógico (o incluso un enumerado) para indicar qué tipo de letras se quiere y llamar a una función que reciba ese parámetro y actúe en consecuencia. Puedes implementarlo así como ejercicio extra.

Ex. 21: Crea un programa que dado un número como argumento imprima una pirámide como esta de tantos pisos como el número indicado.

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

char **make_pyramid(int steps) {
    char **result = malloc(sizeof(*result) * steps);
    for (int ii = 0; ii < steps; ++ii) {
        result[ii] = malloc(sizeof(**result) * (steps * 2));
        memset(result[ii], ' ', steps * 2 - 1);
        memset(result[ii] + ii, '%', (steps * 2 - 1) - 2 * ii);
        result[ii][(steps - 1) * 2 + 1] = 0;
    }
    return result;
}

void free_pyramid(char **pyramid, int steps) {
    for (int ii = 0; ii < steps; ++ii) {
        free(pyramid[ii]);
    }
    free(pyramid);
}

int main(int argc, char const *argv[]) {
    int steps     = atoi(argv[1]);
    char **pyramid = make_pyramid(steps);
    for (int ii = 0; ii < steps; ++ii) {
        printf("%s\n", pyramid[ii]);
    }
    free_pyramid(pyramid, steps);
}
```

Program 189: Solución al ejercicio 21

Ex. 22: Escribe un programa que reciba una serie de puntos y de nombres para cada uno y después los imprima en orden de su distancia al origen de menor a mayor.

```c
1  #ifndef TAGGED_POINT_H
2  #define TAGGED_POINT_H
3  typedef struct tagged_point_s tagged_point_t;
4
5  tagged_point_t *tagged_point_create(const char *tag, double x,
6                                      double y);
7
8  void tagged_point_set_tag(const char *tag, tagged_point_t *tp);
9
10 void tagged_point_set_x(double x, tagged_point_t *tp);
11
12 void tagged_point_set_y(double y, tagged_point_t *tp);
13
14 const char *tagged_point_get_tag(const tagged_point_t *tp);
15
16 double tagged_point_get_x(const tagged_point_t *tp);
17
18 double tagged_point_get_y(const tagged_point_t *tp);
19
20 void tagged_point_destroy(tagged_point_t *tp);
21
22 double tagged_point_distance(const tagged_point_t *a,
23                              const tagged_point_t *b);
24 #endif
```

Program 190: Solución al ejercicio 23 – `tagged_point.h`

```c
1  #include "tagged_point.h"
2  #include <math.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  struct tagged_point_s {
7      char *tag;
8      double x, y;
9  };
10
11 tagged_point_t *tagged_point_create(const char *tag, double x,
12                                     double y) {
13     tagged_point_t *res = malloc(sizeof(tagged_point_t));
14     res->x            = x;
15     res->y            = y;
16     res->tag          = strdup(tag);
17 }
18
19 void tagged_point_set_tag(const char *tag, tagged_point_t *tp) {
20     free(tp->tag);
21     tp->tag = strdup(tag);
22 }
23
24 void tagged_point_set_x(double x, tagged_point_t *tp) { tp->x = x; }
25
26 void tagged_point_set_y(double y, tagged_point_t *tp) { tp->y = y; }
27
28 const char *tagged_point_get_tag(const tagged_point_t *tp) {
29     return tp->tag;
30 }
31
32 double tagged_point_get_x(const tagged_point_t *tp) { return tp->x; }
33
34 double tagged_point_get_y(const tagged_point_t *tp) { return tp->y; }
35
36 void tagged_point_destroy(tagged_point_t *tp) {
37     free(tp->tag);
38     free(tp);
39 }
40
41 double tagged_point_distance(const tagged_point_t *a,
42                             const tagged_point_t *b) {
43     tagged_point_t origin = {"origin", 0.0, 0.0};
44     if (NULL == a) {
45         a = &origin;
46     }
47     if (NULL == b) {
48         b = &origin;
49     }
50     return sqrt((a->x - b->x) * (a->x - b->x) +
51                 (a->y - b->y) * (a->y - b->y));
52 }
```

Program 191: Solución al ejercicio 22 – `tagged_point.c`

```c
1  #include "tagged_point.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  typedef int(comparator_t)(const void *, const void *);
7
8  void generic_swap(void *one, void *other, size_t type_size) {
9  // ...
10
11 void generic_bubble_sort(void *array, size_t array_size,
12 // ...
13
14 int compare_distance(const void *a, const void *b) {
15     tagged_point_t *p1 = *(tagged_point_t **)a;
16     tagged_point_t *p2 = *(tagged_point_t **)b;
17     return tagged_point_distance(NULL, p1) <
18             tagged_point_distance(NULL, p2);
19 }
20
21 int main(int argc, char const *argv[]) {
22
23     int point_lenght = 0;
24     if ((argc - 1) % 3 != 0) {
25         printf("Algo parece estar mal.");
26         return EXIT_FAILURE;
27     }
28     point_lenght = (argc - 1) / 3;
29     tagged_point_t *points[point_lenght];
30     for (int ii = 0; ii < point_lenght; ++ii) {
31         double x       = atof(argv[1 + ii * 3 + 0]);
32         double y       = atof(argv[1 + ii * 3 + 1]);
33         const char *tag = argv[1 + ii * 3 + 2];
34         points[ii]      = tagged_point_create(tag, x, y);
35     }
36
37     generic_bubble_sort(points, point_lenght, sizeof(tagged_point_t *),
38                         compare_distance);
39
40     for (int ii = 0; ii < point_lenght; ++ii) {
41         printf("%f %f %s\n", tagged_point_get_x(points[ii]),
42                 tagged_point_get_y(points[ii]),
43                 tagged_point_get_tag(points[ii]));
44         tagged_point_destroy(points[ii]);
45     }
46 }
```

Program 192: Solución al ejercicio 22 – `main.c`