

**THE
C
PROGRAMMING
LANGUAGE
HANDBOOK**

FLAVIO COPES

Table of Contents

Preface

Introduction to C

Variables and types

Constants

Operators

Conditionals

Loops

Arrays

Strings

Pointers

Functions

Input and output

Variables scope

Static variables

Global variables

Type definitions

Enumerated Types

Structures

Command line parameters

Header files

The preprocessor

Conclusion

Preface

The C Programming Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

I find this approach gives a well-rounded overview.

This book does not try to cover everything under the sun related to C. It focuses on the core of the language, trying to simplify the more complex topics.

I hope the contents of this book will help you achieve what you want: **learn the basics of C**.

This book is written by Flavio. I **publish programming tutorials** every day on my website flaviocopes.com.

You can reach me on Twitter [@flaviocopes](https://twitter.com/flaviocopes).

Enjoy!

Introduction to C

C is probably the most widely known programming language. It is used as the reference language for computer science courses all over the world, and it's probably the language that people learn the most in school among with Python and Java.

I remember it being my second programming language ever, after Pascal.

C is not just what students use to learn programming. It's not an academic language. And I would say it's not the easiest language, because C is a rather low level programming language.

Today, C is widely used in embedded devices, and it powers most of the Internet servers, which are built using Linux. The Linux kernel is built using C, and this also means that C powers the core of all Android devices. We can say that C code runs a good portion of the entire world. Right now. Pretty remarkable.

When it was created, C was considered a high level language, because it was portable across machines. Today we kind of give for granted that we can run a program written on a Mac on Windows or Linux, perhaps using Node.js or Python. Once upon a time, this was not the case at all. What C brought to the table was a language simple to implement, having a compiler that could be easily ported to different machines.

I said compiler: C is a compiled programming language, like Go, Java, Swift or Rust. Other popular programming language like Python, Ruby or JavaScript are interpreted. The difference is consistent: a compiled language generates a binary file that can be directly executed and distributed.

C is not garbage collected. This means we have to manage memory ourselves. It's a complex task and one that requires a lot of attention to prevent bugs, but it is also what makes C ideal to write programs for embedded devices like Arduino.

C does not hide the complexity and the capabilities of the machine underneath. You have a lot of power, once you know what you can do.

I want to introduce the first C program now, which we'll call "Hello, World!"

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}
```

Let's describe the program source code: we first import the `stdio` library (the name stands for standard input-output library).

This library gives us access to input/output functions.

C is a very small language at its core, and anything that's not part of the core is provided by libraries. Some of those libraries are built by normal programmers, and made available for others to use. Some other libraries are built into the compiler. Like `stdio` and others.

`stdio` is the libraries that provides the `printf()` function.

This function is wrapped into a `main()` function. The `main()` function is the entry point of any C program.

But what is a function, anyway?

A function is a routine that takes one or more arguments, and returns a single value.

In the case of `main()`, the function gets no arguments, and returns an integer. We identify that using the `void` keyword for the argument, and the `int` keyword for the return value.

The function has a body, which is wrapped in curly braces, and inside the body we have all the code that the function needs to perform its operations.

The `printf()` function is written differently, as you can see. It has no return value defined, and we pass a string, wrapped in double quotes. We didn't specify the type of the argument.

That's because this is a function invocation. Somewhere, inside the `stdio` library, `printf` is defined as

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

You don't need to understand what this means now, but in short, this is the definition and when we call `printf("Hello, World!");`, that's where the function is ran.

The `main()` function we defined above:

```
#include <stdio.h>

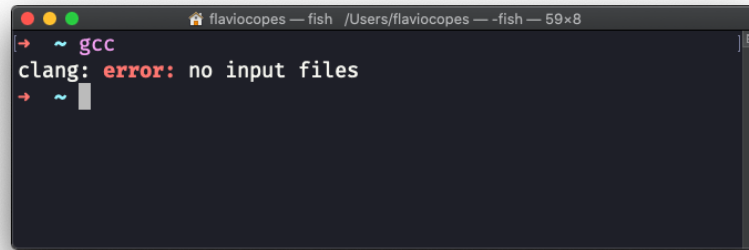
int main(void) {
    printf("Hello, World!");
}
```

will be ran by the operating system when the program is executed.

How do we execute a C program?

As mentioned, C is a compiled language. To run the program we must first compile it. Any Linux or macOS computer already comes with a C compiler built-in. For Windows, you can use the Windows Subsystem for Linux (WSL).

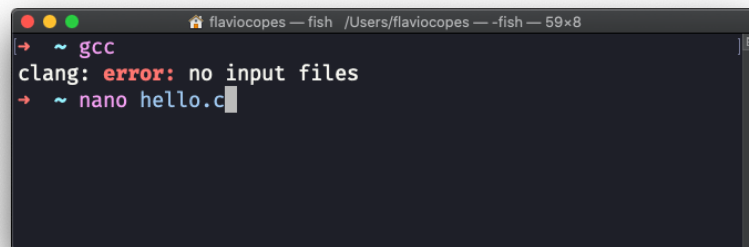
In any case, when you open the terminal window you can type `gcc`, and this command should return you an error saying that you didn't specify any file:



```
flaviocopes — fish /Users/flaviocopes — -fish — 59x8
~ gcc
clang: error: no input files
~
```

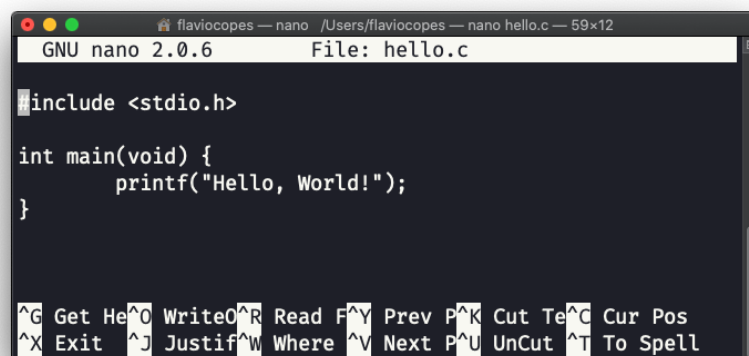
That's good. It means the C compiler is there, and we can start using it.

Now type the program above into a `hello.c` file. You can use any editor, but for the sake of simplicity I'm going to use the `nano` editor in the command line:



```
flaviocopes — fish /Users/flaviocopes — -fish — 59x8
~ gcc
clang: error: no input files
~ nano hello.c
```

Type the program:



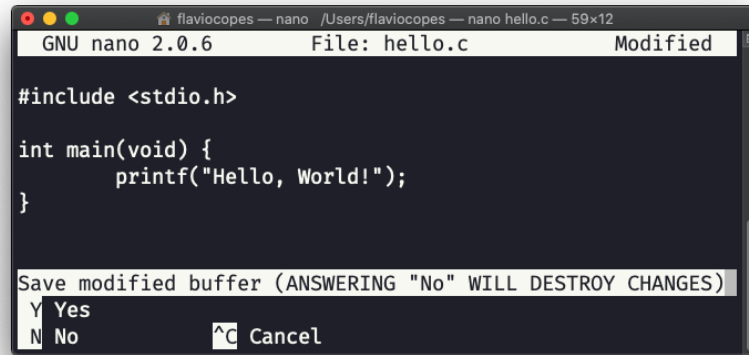
```
flaviocopes — nano /Users/flaviocopes — nano hello.c — 59x12
GNU nano 2.0.6 File: hello.c

#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}

^G Get He ^O WriteO ^R Read F ^Y Prev P ^K Cut Te ^C Cur Pos
^X Exit ^J Justif ^W Where ^V Next P ^U UnCut ^T To Spell
```

Now press `ctrl-X` to exit:



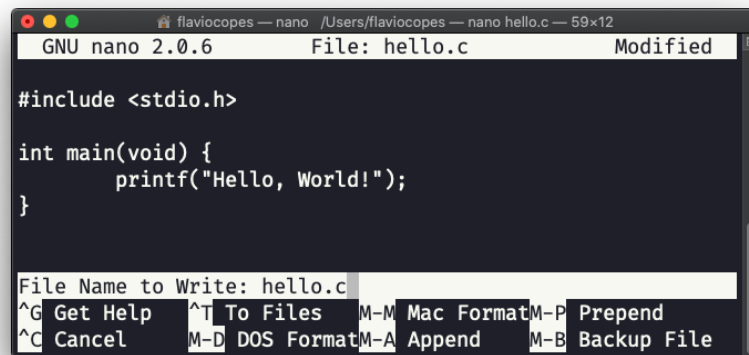
```
GNU nano 2.0.6 File: hello.c Modified

#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}

Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES)
Y Yes
N No ^C Cancel
```

Confirm by pressing the `y` key, then press enter to confirm the file name:



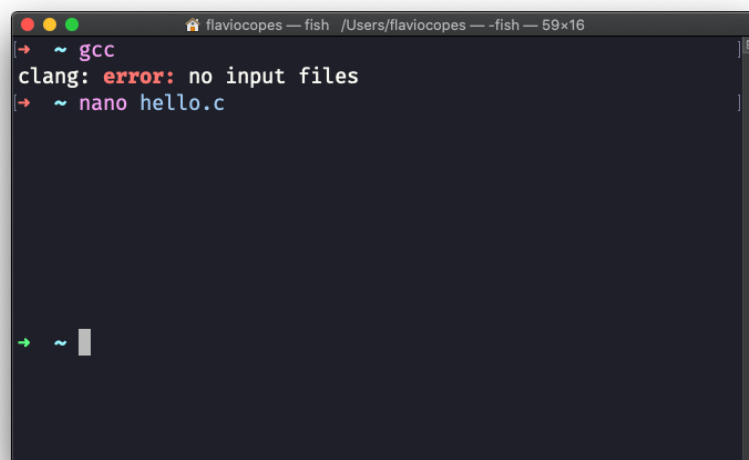
```
GNU nano 2.0.6 File: hello.c Modified

#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}

File Name to Write: hello.c
^G Get Help ^T To Files M-M Mac Format M-P Prepend
^C Cancel M-D DOS Format M-A Append M-B Backup File
```

That's it, we should be back to the terminal now:



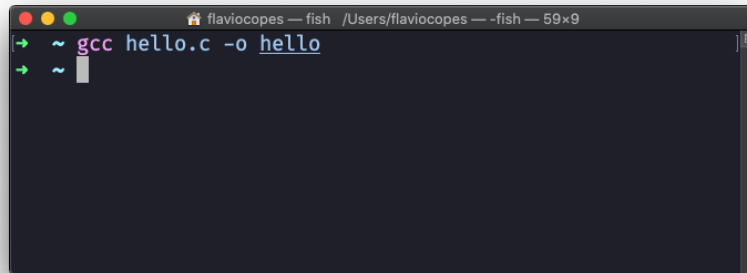
```
flaviocopes — fish /Users/flaviocopes — -fish — 59x16
[→ ~ gcc
clang: error: no input files
[→ ~ nano hello.c

[→ ~ ]
```

Now type

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

The program should give you no errors:

A terminal window with a dark background and light-colored text. The title bar shows 'flaviocopes — fish /Users/flaviocopes — -fish — 59x9'. The prompt is '~'. The command 'gcc hello.c -o hello' has been entered and executed. The prompt is now '~' again, with a cursor at the end.

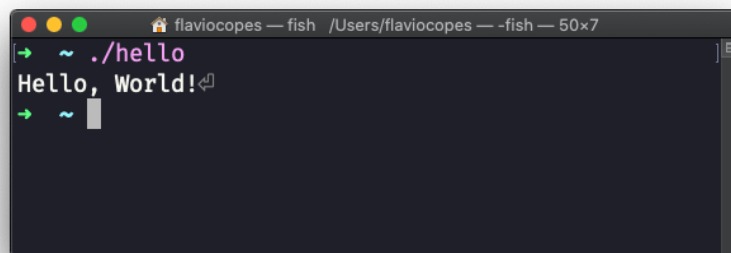
```
flaviocopes — fish /Users/flaviocopes — -fish — 59x9
~ gcc hello.c -o hello
~
```

but it should have generated a `hello` executable.

Now type

```
./hello
```

to run it:

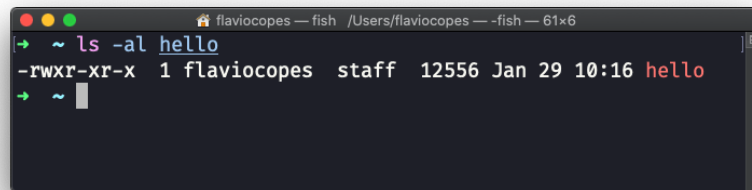
A terminal window with a dark background and light-colored text. The title bar shows 'flaviocopes — fish /Users/flaviocopes — -fish — 50x7'. The prompt is '~'. The command './hello' has been entered and executed. The output 'Hello, World!' is displayed. The prompt is now '~' again, with a cursor at the end.

```
flaviocopes — fish /Users/flaviocopes — -fish — 50x7
~ ./hello
Hello, World!
~
```

I prepend `./` to the program name, to tell the terminal that the command is in the current folder

Awesome!

Now if you call `ls -al hello`, you can see that the program is only 12KB in size:



```
flaviocopes — fish /Users/flaviocopes — -fish — 61x6
→ ~ ls -al hello
-rwxr-xr-x 1 flaviocopes staff 12556 Jan 29 10:16 hello
→ ~
```

This is one of the pros of C: it's highly optimized, and this is also one of the reasons it's this good for embedded devices that have a very limited amount of resources.

Variables and types

C is a statically typed language.

This means that any variable has an associated type, and this type is known at compilation time.

This is very different than how you work with variables in Python, JavaScript, PHP and other interpreted languages.

When you create a variable in C, you have to specify the type of a variable at the declaration.

In this example we initialize a variable `age` with type `int` :

```
int age;
```

A variable name can contain any uppercase or lowercase letter, can contain digits and the underscore character, but it can't start with a digit. `AGE` and `Age10` are valid variable names, `1age` is not.

You can also initialize a variable at declaration, specifying the initial value:

```
int age = 37;
```

Once you declare a variable, you are then able to use it in your program code, and you can change its value at any time, using the `=` operator for example, like in `age = 100;` , provided the new value is of the same type.

In this case:

```
#include <stdio.h>

int main(void) {
    int age = 0;
    age = 37.2;
    printf("%u", age);
}
```

the compiler will raise a warning at compile time, and will convert the decimal number to an integer value.

The C built-in data types are `int` , `char` , `short` , `long` , `float` , `double` , `long double` . Let's find out more about those.

Integer numbers

C provides us the following types to define integer values:

- `char`
- `int`
- `short`
- `long`

Most of the times, you'll likely use an `int` to store an integer. But in some cases, you might want to choose one of the other 3 options.

The `char` type is commonly used to store letters of the ASCII chart, but it can be used to hold small integers from `-128` to `127` . It takes at least 1 byte.

`int` takes at least 2 bytes. `short` takes at least 2 bytes. `long` takes at least 4 bytes.

As you can see, we are not guaranteed the same values for different environments. We only have an indication. The problem is that the exact numbers that can be stored in each data type depends on the implementation and the architecture.

We're guaranteed that `short` is not longer than `int` . And we're guaranteed `long` is not shorter than `int` .

The ANSI C spec standard determines the minimum values of each type, and thanks to it we can at least know what's the minimum value we can expect to have at our disposal.

If you are programming C on an Arduino, different board will have different limits.

On an Arduino Uno board, `int` stores a 2 byte value, ranging from `-32,768` to `32,767` . On a Arduino MKR 1010, `int` stores a 4 bytes value, ranging from `-2,147,483,648` to `2,147,483,647` . Quite a big difference.

On all Arduino boards, `short` stores a 2 bytes value, ranging from `-32,768` to `32,767` . `long` store 4 bytes, ranging from `-2,147,483,648` to `2,147,483,647` .

Unsigned integers

For all the above data types, we can prepend `unsigned` to start the range at 0, instead of a negative number. This might make sense in many cases.

- `unsigned char` will range from `0` to at least `255`
- `unsigned int` will range from `0` to at least `65,535`
- `unsigned short` will range from `0` to at least `65,535`
- `unsigned long` will range from `0` to at least `4,294,967,295`

The problem with overflow

Given all those limits, a question might come up: how can we make sure our numbers do not exceed the limit? And what happens if we do exceed the limit?

If you have an `unsigned int` number at 255, and you increment it, you'll get 256 in return. As expected. If you have a `unsigned char` number at 255, and you increment it, you'll get 0 in return. It resets starting from the initial possible value.

If you have a `unsigned char` number at 255 and you add 10 to it, you'll get the number `9` :

```
#include <stdio.h>

int main(void) {
    unsigned char j = 255;
    j = j + 10;
    printf("%u", j); /* 9 */
}
```

If you don't have a signed value, the behavior is undefined. It will basically give you a huge number which can vary, like in this case:

```
#include <stdio.h>

int main(void) {
    char j = 127;
    j = j + 10;
    printf("%u", j); /* 4294967177 */
}
```

In other words, C does not protect you from going over the limits of a type. You need to take care of this yourself.

Warnings when declaring the wrong type

When you declare the variable and initialize it with the wrong value, the `gcc` compiler (the one you're probably using) should warn you:

```
#include <stdio.h>

int main(void) {
    char j = 1000;
}
```



```
hello.c:4:11: warning: implicit conversion
    from 'int' to
      'char' changes value from 1000 to -24
    [-Wconstant-conversion]
    char j = 1000;
           ~  ^~~~
1 warning generated.
```

And it also warns you in direct assignments:

```
#include <stdio.h>

int main(void) {
    char j;
    j = 1000;
}
```

But not if you increase the number using for example

`+=` :

```
#include <stdio.h>

int main(void) {
    char j = 0;
    j += 1000;
}
```

Floating point numbers

Floating point types can represent a much larger set of values than integers can, and can also represent fractions, something that integers can't do.

Using floating point numbers, we represent numbers as decimal numbers times powers of 10.

You might see floating point numbers written as

- `1.29e-3`
- `-2.3e+5`

and in other seemingly weird ways.

The following types:

- `float`
- `double`
- `long double`

are used to represent numbers with decimal points (floating point types). All can represent both positive and negative numbers.

The minimum requirements for any C implementation is that `float` can represent a range between 10^{-37} and 10^{+37} , and is typically implemented using 32 bits. `double` can represent a bigger set of numbers. `long double` can hold even more numbers.

The exact figures, as with integer values, depend on the implementation.

On a modern Mac, a `float` is represented in 32 bits, and has a precision of 24 significant bits, 8 bits are used to encode the exponent. A `double` number is represented in 64 bits, with a precision of 53 significant bits, 11 bits are used to encode the exponent. The type `long double` is represented in 80 bits, has a precision of 64 significant bits, 15 bits are used to encode the exponent.

On your specific computer, how can you determine the specific size of the types? You can write a program to do that:

```
#include <stdio.h>

int main(void) {
    printf("char size: %lu bytes\n", sizeof(char));
    printf("int size: %lu bytes\n", sizeof(int));
    printf("short size: %lu bytes\n", sizeof(short));
    printf("long size: %lu bytes\n", sizeof(long));
    printf("float size: %lu bytes\n", sizeof(float));
    printf("double size: %lu bytes\n",
           sizeof(double));
    printf("long double size: %lu bytes\n",
           sizeof(long double));
}
```

In my system, a modern Mac, it prints:

```
char size: 1 bytes
int size: 4 bytes
short size: 2 bytes
long size: 8 bytes
float size: 4 bytes
double size: 8 bytes
long double size: 16 bytes
```

Constants

In the last post I introduced [variables in C](#).

In this post I want to tell you everything about constants in C.

A constant is declared similarly to variables, except it is prepended with the `const` keyword, and you always need to specify a value.

Like this:

```
const int age = 37;
```

This is perfectly valid C, although it is common to declare constants uppercase, like this:

```
const int AGE = 37;
```

It's just a convention, but one that can greatly help you while reading or writing a C program as it improves readability. Uppercase name means constant, lowercase name means variable.

A constant name follows the same rules for variable names: can contain any uppercase or lowercase letter, can contain digits and the underscore character, but it can't start with a digit. `AGE` and `Age10` are valid variable names, `1AGE` is not.

Another way to define constants is by using this syntax:

```
#define AGE 37
```

In this case, you don't need to add a type, and you don't also need the `=` equal sign, and you omit the semicolon at the end.

The C compiler will infer the type from the value specified, at compile time.

Operators

C offers us a wide variety of operators that we can use to operate on data.

In particular, we can identify various groups of operators:

- arithmetic operators
- comparison operators
- logical operators
- compound assignment operators
- bitwise operators
- pointer operators
- structure operators
- miscellaneous operators

In this blog post I'm going to detail all of them, using 2 imaginary variables `a` and `b` as examples.

I am keeping bitwise operators, structure operators and pointer operators out of this list, as I will dedicate them a specific blog post.

Arithmetic operators

In this macro group I am going to separate binary operators and unary operators.

Binary operators work using two operands:

Operator	Name	Example
=	Assignment	a = b
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulo	a % b

Unary operators only take one operand:

Operator	Name	Example
+	Unary plus	+a
-	Unary minus	-a
++	Increment	a++ or ++a
--	Decrement	a-- or --a

The difference between `a++` and `++a` is that `a++` increments the `a` variable after using it. `++a` increments the `a` variable before using it.

For example:

```
int a = 2;
int b;
b = a++ /* b is 2, a is 3 */
b = ++a /* b is 4, a is 4 */
```

The same applies to the decrement operator.

Comparison operators

Operator	Name	Example
<code>==</code>	Equal operator	<code>a == b</code>
<code>!=</code>	Not equal operator	<code>a != b</code>
<code>></code>	Bigger than	<code>a > b</code>
<code><</code>	Less than	<code>a < b</code>
<code>>=</code>	Bigger than or equal to	<code>a >= b</code>
<code><=</code>	Less than or equal to	<code>a <= b</code>

Logical operators

- `!` NOT (example: `!a`)
- `&&` AND (example: `a && b`)
- `||` OR (example: `a || b`)

Those operators are great when working with boolean values.

Compound assignment operators

Those operators are useful to perform an assignment and at the same time perform an arithmetic operation:

Operator	Name	Example
<code>+=</code>	Addition assignment	<code>a += b</code>
<code>-=</code>	Subtraction assignment	<code>a -= b</code>
<code>*=</code>	Multiplication assignment	<code>a *= b</code>
<code>/=</code>	Division assignment	<code>a /= b</code>
<code>%=</code>	Modulo assignment	<code>a %= b</code>

Miscellaneous operators

The ternary operator

The ternary operator is the only operator in `C` that works with 3 operands, and it's a short way to express conditionals.

This is how it looks:

```
<condition> ? <expression> : <expression>
```

Example:

```
a ? b : c
```

If `a` is evaluated to `true`, then the `b` statement is executed, otherwise `c` is.

The ternary operator is functionality-wise same as an if/else conditional, except it is shorter to express and it can be inlined into an expression.

sizeof

The `sizeof` operator returns the size of the operand you pass. You can pass a variable, or even a type.

Example usage:

```
#include <stdio.h>

int main(void) {
    int age = 37;
    printf("%ld\n", sizeof(age));
    printf("%ld", sizeof(int));
}
```

Operator precedence

With all those operators (and more, which I haven't covered in this post, including bitwise, structure operators and pointer operators), we must pay attention when using them together in a single expression.

Suppose we have this operation:

```
int a = 2;
int b = 4;
int c = b + a * a / b - a;
```

What's the value of `c` ? Do we get the addition being executed before the multiplication and the division?

There is a set of rules that help us solving this puzzle.

In order from less precedence to more precedence, we have:

- the `=` assignment operator
- the `+` and `-` **binary** operators
- the `*` and `/` operators
- the `+` and `-` unary operators

Operators also have an associativity rule, which is always left to right except for the unary operators and the assignment.

In:

```
int c = b + a * a / b - a;
```

We first execute `a * a / b`, which due to being left-to-right we can separate into `a * a` and the result `/ b`: `2 * 2 = 4`, `4 / 4 = 1`.

Then we can perform the sum and the subtraction: `4 + 1 - 2`. The value of `c` is `3`.

In all cases, however, I want to make sure you realize you can use parentheses to make any similar expression easier to read and comprehend.

Parentheses have higher priority over anything else.

The above example expression can be rewritten as:

```
int c = b + ((a * a) / b) - a;
```

and we don't have to think about it that much.

Conditionals

Any programming language provides the programmers the ability to perform choices.

We want to do X in some cases, and Y in other cases.

We want to check data, and do choices based on the state of that data.

C provides us 2 ways to do so.

The first is the `if` statement, with its `else` helper, and the second is the `switch` statement.

`if`

In an `if` statement, you can check for a condition to be true, and then execute the block provided in the curly brackets:

```
int a = 1;

if (a == 1) {
    /* do something */
}
```

You can append an `else` block to execute a different block if the original condition turns out to be false;

```
int a = 1;

if (a == 2) {
    /* do something */
} else {
    /* do something else */
}
```

Beware one common source of bugs - always use the comparison operator `==` in comparisons, and not the assignment operator `=`, otherwise the `if` conditional check will always be true, unless the argument is `0`, for example if you do:

```
int a = 0;

if (a = 0) {
    /* never invoked */
}
```

Why does this happen? Because the conditional check will look for a boolean result (the result of a comparison), and the `0` number always equates to a false value. Everything else is true, including negative numbers.

You can have multiple `else` blocks by stacking together multiple `if` statements:

```
int a = 1;

if (a == 2) {
    /* do something */
} else if (a == 1) {
    /* do something else */
} else {
    /* do something else again */
}
```

switch

If you need to do too many if / else / if blocks to perform a check, perhaps because you need to check the exact value of a variable, then `switch` can be very useful to you.

You can provide a variable as condition, and a series of `case` entry points for each value you expect:

```
int a = 1;

switch (a) {
    case 0:
        /* do something */
        break;
    case 1:
        /* do something else */
        break;
    case 2:
        /* do something else */
        break;
}
```

We need a `break` keyword at the end of each case, to avoid the next case to be executed when the one before ends. This "cascade" effect can be useful in

some creative ways.

You can add a "catch-all" case at the end, labeled `default :`

```
int a = 1;

switch (a) {
    case 0:
        /* do something */
        break;
    case 1:
        /* do something else */
        break;
    case 2:
        /* do something else */
        break;
    default:
        /* handle all the other cases */
        break;
}
```

Loops

C offers us three ways to perform a loop: **for loops**, **while loops** and **do while loops**. They all allow you to iterate over arrays, but with a few differences. Let's see them in details.

For loops

The first, and probably most common, way to perform a loop is **for loops**.

Using the `for` keyword we can define the *rules* of the loop up front, and then provide the block that is going to be executed repeatedly.

Like this:

```
for (int i = 0; i <= 10; i++) {  
    /* instructions to be repeated */  
}
```

The `(int i = 0; i <= 10; i++)` block contains 3 parts of the looping details:

- the initial condition (`int i = 0`)
- the test (`i <= 10`)
- the increment (`i++`)

We first define a loop variable, in this case named `i`. `i` is a common variable name to be used for loops, along with `j` for nested loops (a loop inside another loop). It's just a convention.

The variable is initialized at the 0 value, and the first iteration is done. Then it is incremented as the increment part says (`i++` in this case, incrementing by 1), and all the cycle repeats until you get to the number 10.

Inside the loop main block we can access the variable `i` to know at which iteration we are. This program should print `0 1 2 3 4 5 5 6 7 8 9 10 :`

```
for (int i = 0; i <= 10; i++) {  
    /* instructions to be repeated */  
    printf("%u ", i);  
}
```

Loops can also start from a high number, and go a lower number, like this:

```
for (int i = 10; i > 0; i--) {  
    /* instructions to be repeated */  
}
```

You can also increment the loop variable by 2 or another value:

```
for (int i = 0; i < 1000; i = i + 30) {  
    /* instructions to be repeated */  
}
```

While loops

While loops is simpler to write than a `for` loop, because it requires a bit more work on your part.

Instead of defining all the loop data up front when you start the loop, like you do in the `for` loop, using `while` you just check for a condition:

```
while (i < 10) {  
  
}
```

This assumes that `i` is already defined and initialized with a value.

And this loop will be an **infinite loop** unless you increment the `i` variable at some point inside the loop. An infinite loop is bad because it will block the program, nothing else can happen.

This is what you need for a "correct" while loop:

```
int i = 0;  
  
while (i < 10) {  
    /* do something */  
  
    i++;  
}
```

There's one exception to this, and we'll see it in one minute. Before, let me introduce `do while`.

Do while loops

While loops are great, but there might be times when you need to do one particular thing: you want to always execute a block, and then *maybe* repeat it.

This is done using the `do while` keyword, in a way that's very similar to a `while` loop, but slightly different:

```
int i = 0;

do {
    /* do something */

    i++;
} while (i < 10);
```

The block that contains the `/* do something */` comment is always executed at least once, regardless of the condition check at the bottom.

Then, until `i` is less than 10, we'll repeat the block.

Breaking out of a loop using `break`

In all the `C` loops we have a way to break out of a loop at any point in time, immediately, regardless of the conditions set for the loop.

This is done using the `break` keyword.

This is useful in many cases. You might want to check for the value of a variable, for example:

```
for (int i = 0; i <= 10; i++) {
    if (i == 4 && someVariable == 10) {
        break;
    }
}
```

Having this option to break out of a loop is particularly interesting for `while` loops (and `do while` too), because we can create seemingly infinite loops that end when a condition occurs, and you define this inside the loop block:

```
int i = 0;
while (1) {
    /* do something */

    i++;
    if (i == 10) break;
}
```

It's rather common to have this kind of loops in C.

Arrays

An array is a variable that stores multiple values.

Every value in the array, in C, must have the **same type**. This means you will have arrays of `int` values, arrays of `double` values, and more.

You can define an array of `int` values like this:

```
int prices[5];
```

You must always specify the size of the array. C does not provide dynamic arrays out of the box (you have to use a data structure like a linked list for that).

You can use a constant to define the size:

```
const int SIZE = 5;  
int prices[SIZE];
```

You can initialize an array at definition time, like this:

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

But you can also assign a value after the definition, in this way:

```
int prices[5];

prices[0] = 1;
prices[1] = 2;
prices[2] = 3;
prices[3] = 4;
prices[4] = 5;
```

Or, more practical, using a loop:

```
int prices[5];

for (int i = 0; i < 5; i++) {
    prices[i] = i + 1;
}
```

And you can reference an item in the array by using square brackets after the array variable name, adding an integer to determine the index value. Like this:

```
prices[0]; /* array item value: 1 */
prices[1]; /* array item value: 2 */
```

Array indexes start from 0, so an array with 5 items, like the `prices` array above, will have items ranging from `prices[0]` to `prices[4]` .

The interesting thing about C arrays is that all elements of an array are stored sequentially, one right after another. Not something that normally happens with higher-level programming languages.

Another interesting thing is this: the variable name of the array, `prices` in the above example, is a **pointer** to the first element of the array, and as such can be

used like a normal pointer.

More on pointers soon.

Strings

In C, strings are one special kind of array: a string is an array of `char` values:

```
char name[7];
```

I introduced the `char` type when I introduced types, but in short it is commonly used to store letters of the ASCII chart.

A string can be initialized like you initialize a normal array:

```
char name[7] = { "F", "l", "a", "v", "i", "o" };
```

Or more conveniently with a string literal (also called string constant), a sequence of characters enclosed in double quotes:

```
char name[7] = "Flavio";
```

You can print a string via `printf()` using `%s` :

```
printf("%s", name);
```

Do you notice how "Flavio" is 6 chars long, but I defined an array of length 7? Why? This is because the last character in a string must be a `0` value, the string terminator, and we must make space for it.

This is important to keep in mind especially when manipulating strings.

Speaking of manipulating strings, there's one important standard library that is provided by C: `string.h`.

This library is essential because it abstracts many of the low level details of working with strings, and provides us a set of useful functions.

You can load the library in your program by adding on top:

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

And once you do that, you have access to:

- `strcpy()` to copy a string over another string
- `strcat()` to append a string to another string
- `strcmp()` to compare two strings for equality
- `strncmp()` to compare the first `n` characters of two strings
- `strlen()` to calculate the length of a string

and many, many more.

I will introduce all those string functions in separate blog posts, but just know that they exist.

Pointers

Pointers are one of the most confusing/challenging parts of C, in my opinion. Especially if you are new to programming, but also if you come from a higher level programming language like Python or JavaScript.

In this post I want to introduce them in the simplest yet not-dumbed-down way possible.

A pointer is the address of a block of memory that contains a variable.

When you declare an integer number like this:

```
int age = 37;
```

We can use the `&` operator to get the value of the address in memory of a variable:

```
printf("%p", &age); /* 0x7ffeef7dcb9c */
```

I used the `%p` format specified in `printf()` to print the address value.

We can assign the address to a variable:

```
int *address = &age;
```

Using `int *address` in the declaration, we are not declaring an integer variable, but rather a **pointer to an integer**.

We can use the pointer operator `*` to get the value of the variable an address is pointing to:

```
int age = 37;
int *address = &age;
printf("%u", *address); /* 37 */
```

This time we are using the pointer operator again, but since it's not a declaration this time it means "the value of the variable this pointer points to".

In this example we declare an `age` variable, and we use a pointer to initialize the value:

```
int age;
int *address = &age;
*address = 37;
printf("%u", *address);
```

When working with C, you'll find that a lot of things are built on top of this simple concept, so make sure you familiarize with it a bit, by running the above examples on your own.

Pointers are a great opportunity because they force us to think about memory addresses and how data is organized.

Arrays are one example. When you declare an array:

```
int prices[3] = { 5, 4, 3 };
```

The `prices` variable is actually a pointer to the first item of the array. You can get the value of the first item using this `printf()` function in this case:

```
printf("%u", *prices); /* 5 */
```

The cool thing is that we can get the second item by adding 1 to the `prices` pointer:

```
printf("%u", *(prices + 1)); /* 4 */
```

And so on for all the other values.

We can also do many nice string manipulation operations, since strings are arrays under the hood.

We also have many more applications, including passing the reference of an object or a function around, to avoid consuming more resources to copy it.

Functions

Functions are the way we can structure our code into subroutines that we can:

1. give a name to
2. call when we need them

Starting from your very first program, an "Hello, World!", you immediately make use of C functions:

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}
```

The `main()` function is a very important function, as it's the entry point for a C program.

Here's another function:

```
void doSomething(int value) {
    printf("%u", value);
}
```

Functions have 4 important aspects:

1. they have a name, so we can invoke ("call") them later
2. they specify a return value
3. they can have arguments
4. they have a body, wrapped in curly braces

The function body is the set of instructions that are executed any time we invoke a function.

If the function has no return value, you can use the keyword `void` before the function name. Otherwise you specify the function return value type (`int` for an integer, `float` for a floating point value, `const char *` for a string, etc).

You cannot return more than one value from a function.

A function can have arguments. They are optional. If it does not have them, inside the parentheses we insert `void`, like this:

```
void doSomething(void) {  
    /* ... */  
}
```

In this case, when we invoke the function we'll call it with nothing in the parentheses:

```
doSomething();
```

If we have one parameter, we specify the type and the name of the parameter, like this:

```
void doSomething(int value) {  
    /* ... */  
}
```

When we invoke the function, we'll pass that parameter in the parentheses, like this:

```
doSomething(3);
```

We can have multiple parameters, and if so we separate them using a comma, both in the declaration and in the invocation:

```
void doSomething(int value1, int value2) {  
    /* ... */  
}  
  
doSomething(3, 4);
```

Parameters are passed by **copy**. This means that if you modify `value1`, its value is modified locally, and the value outside of the function, where it was passed in the invocation, does not change.

If you pass a **pointer** as a parameter, you can modify that variable value because you can now access it directly using its memory address.

You can't define a default value for a parameter. C++ can do that (and so Arduino Language programs can), but C can't.

Make sure you define the function before calling it, or the compiler will raise a warning and an error:

```

→ ~ gcc hello.c -o hello; ./hello
hello.c:13:3: warning: implicit declaration of
      function 'doSomething' is invalid in C99
      [-Wimplicit-function-declaration]
      doSomething(3, 4);
      ^
hello.c:17:6: error: conflicting types for
      'doSomething'
void doSomething(int value1, char value2) {
      ^
hello.c:13:3: note: previous implicit declaration
      is here
      doSomething(3, 4);
      ^
1 warning and 1 error generated.

```

The warning you get regards the ordering, which I already mentioned.

The error is about another thing, related. Since C does not "see" the function declaration before the invocation, it must make assumptions. And it assumes the function to return `int`. The function however returns `void`, hence the error.

If you change the function definition to:

```

int doSomething(int value1, int value2) {
    printf("%d %d\n", value1, value2);
    return 1;
}

```

you'd just get the warning, and not the error:


```
→ ~ gcc hello.c -o hello; ./hello
hello.c:14:3: warning: implicit declaration of
      function 'doSomething' is invalid in C99
      [-Wimplicit-function-declaration]
      doSomething(3, 4);
      ^
1 warning generated.
```

In any case, make sure you declare the function before using it. Either move the function up, or add the function prototype in a header file.

Inside a function, you can declare variables.

```
void doSomething(int value) {
    int doubleValue = value * 2;
}
```

A variable is created at the point of invocation of the function, and is destroyed when the function ends, and it's not visible from the outside.

Inside a function, you can call the function itself. This is called **recursion** and it's something that offers peculiar opportunities.

Input and output

C is a small language, and the "core" of C does not include any Input/Output (I/O) functionality.

This is not something unique to C, of course. It's common for the language core to be agnostic of I/O.

In the case of C, Input/Output is provided to us by the C Standard Library via a set of functions defined in the `stdio.h` header file.

You can import this library using:

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

on top of your C file.

This library provides us, among many other functions:

- `printf()`
- `scanf()`
- `sscanf()`
- `fgets()`
- `fprintf()`

Before describing what those functions do, I want to take a minute to talk about **I/O streams**.

We have 3 kinds of I/O streams in C:

- `stdin` (standard input)
- `stdout` (standard output)
- `stderr` (standard error)

With I/O functions we always work with streams. A stream is a high level interface that can represent a device or a file. From the C standpoint, we don't have any difference in reading from a file or reading from the command line: it's an I/O stream in any case.

That's one thing to keep in mind.

Some functions are designed to work with a specific stream, like `printf()`, which we use to print characters to `stdout`. Using its more general counterpart `fprintf()`, we can specify the stream to write to.

Since I started talking about `printf()`, let's introduce it now.

`printf()`

`printf()` is one of the first functions you'll use when learning C programming.

In its simplest usage form, you pass it a string literal:

```
printf("hey!");
```

and the program will print the content of the string to the screen.

You can print the value of a variable, and it's a bit tricky because you need to add a special character, a placeholder, which changes depending on the type of the variable. For example we use `%d` for a signed decimal integer digit:

```
int age = 37;

printf("My age is %d", age);
```

We can print more than one variable by using commas:

```
int age_yesterday = 37;
int age_today = 36;

printf("Yesterday my age was %d and today is %d", age_yesterday, age_today);
```

There are other format specifiers like `%d` :

- `%c` for a char
- `%s` for a string
- `%f` for floating point numbers
- `%p` for pointers

and many more.

We can use escape characters in `printf()` , like `\n` which we can use to make the output create a new line.

scanf()

`printf()` is used as an output function. I want to introduce an input function now, so we can say we can do all the I/O thing: `scanf()` .

This function is used to get a value from the user running the program, from the command line.

We must first define a variable that will hold the value we get from the input:

```
int age;
```

Then we call `scanf()` with 2 arguments: the format (type) of the variable, and the address of the variable:

```
scanf("%d", &age);
```

If we want to get a string as input, remember that a string name is a pointer to the first character, so you don't need the `&` character before it:

```
char name[20];  
scanf("%s", name);
```

Here's a little program that uses both `printf()` and `scanf()` :

```
#include <stdio.h>  
  
int main(void) {  
    char name[20];  
    printf("Enter your name: ");  
    scanf("%s", name);  
    printf("you entered %s", name);  
}
```

Variables scope

When you define a variable in a C program, depending on where you declare it, it will have a different **scope**.

This means that it will be available in some places, but not in others.

The position determines 2 types of variables:

- **global variables**
- **local variables**

This is the difference: a variable declared inside a function is a local variable, like this:

```
int main(void) {  
    int age = 37;  
}
```

Local variables are only accessible from within the function, and when the function ends they stop their existence. They are cleared from the memory (with some exceptions).

A variable defined outside of a function is a global variable, like in this example:

```
int age = 37;  
  
int main(void) {  
    /* ... */  
}
```

Global variables are accessible from any function of the program, and they are available for the whole execution of the program, until it ends.

I mentioned that local variables are not available any more after the function ends.

The reason is that local variables are declared on the **stack**, by default, unless you explicitly allocate them on the heap using pointers, but then you have to manage the memory yourself.

Static variables

Inside a function, you can initialize a **static variable** using the `static` keyword.

I said "inside a function", because global variables are static by default, so there's no need to add the keyword.

What's a static variable? A static variable is initialized to 0 if no initial value is specified, and it retains the value across function calls.

Consider this function:

```
int incrementAge() {  
    int age = 0;  
    age++;  
    return age;  
}
```

If we call `incrementAge()` once, we'll get `1` as the return value. If we call it more than once, we'll always get 1 back, because `age` is a local variable and it's re-initialized to `0` on every single function call.

If we change the function to:

```
int incrementAge() {  
    static int age = 0;  
    age++;  
    return age;  
}
```


Now every time we call this function, we'll get an incremented value:

```
printf("%d\n", incrementAge());  
printf("%d\n", incrementAge());  
printf("%d\n", incrementAge());
```

will give us

```
1  
2  
3
```

We can also omit initializing `age` to 0 in `static int age = 0;`, and just write `static int age;` because static variables are automatically set to 0 when created.

We can also have static arrays. In this case, each single item in the array is initialized to 0:

```
int incrementAge() {  
    static int ages[3];  
    ages[0]++;  
    return ages[0];  
}
```

Global variables

In the [C variables and types](#) post I introduced how to work with variables.

In this post I want to mention the difference between **global and local variables**.

A **local variable** is defined inside a function, and it's only available inside that function.

Like this:

```
#include <stdio.h>

int main(void) {
    char j = 0;
    j += 10;
    printf("%u", j); //10
}
```

`j` is not available anywhere outside the `main` function.

A **global variable** is defined outside of any function, like this:

```
#include <stdio.h>

char i = 0;

int main(void) {
    i += 10;
    printf("%u", i); //10
}
```

A global variable can be accessed by any function in the program. Access is not limited to reading the value: the variable can be updated by any function.

Due to this, global variables are one way we have of sharing the same data between functions.

The main difference with local variables is that the memory allocated for variables is freed once the function ends.

Global variables are only freed when the program ends.

Type definitions

The `typedef` keyword in C allows you to defined new types.

Starting from the [built-in C types](#), we can create our own types, using this syntax:

```
typedef existingtype NEWTYPE
```

The new type we create is usually, by convention, uppercase.

This it to distinguish it more easily, and immediately recognize it as type.

For example we can define a new `NUMBER` type that is an `int` :

```
typedef int NUMBER
```

and once you do so, you can define new `NUMBER` variables:

```
NUMBER one = 1;
```

Now you might ask: why? Why not just use the built-in type `int` instead?

Well, `typedef` gets really useful when paired with two things: enumerated types and structures.

Enumerated Types

An introduction to C Enumerated Types

Using the `typedef` and `enum` keywords we can define a type that can have either one value or another.

It's one of the most important uses of the `typedef` keyword.

This is the syntax of an enumerated type:

```
typedef enum {  
    //...values  
} TYPENAME;
```

The enumerated type we create is usually, by convention, uppercase.

Here is a simple example:

```
typedef enum {  
    true,  
    false  
} BOOLEAN;
```

C comes with a `bool` type, so this example is not really practical, but you get the idea.

Another example is to define weekdays:

```
typedef enum {  
    monday,  
    tuesday,  
    wednesday,  
    thursday,  
    friday,  
    saturday,  
    sunday  
} WEEKDAY;
```

Here's a simple program that uses this enumerated type:

```
#include <stdio.h>  
  
typedef enum {  
    monday,  
    tuesday,  
    wednesday,  
    thursday,  
    friday,  
    saturday,  
    sunday  
} WEEKDAY;  
  
int main(void) {  
    WEEKDAY day = monday;  
  
    if (day == monday) {  
        printf("It's monday!");  
    } else {  
        printf("It's not monday");  
    }  
}
```

Every item in the enum definition is paired to an integer, internally. So in this example `monday` is 0, `tuesday` is 1 and so on.

This means the conditional could have been `if (day == 0)` instead of `if (day == monday)` , but it's way simpler for us humans to reason with names rather than numbers, so it's a very convenient syntax.

Structures

Using the `struct` keyword we can create complex data structures using basic C types.

A structure is a collection of values of different types. Arrays in C are limited to a type, so structures can prove to be very interesting in a lot of use cases.

This is the syntax of a structure:

```
struct <structname> {  
    //...variables  
};
```

Example:

```
struct person {  
    int age;  
    char *name;  
};
```

You can declare variables that have as type that structure by adding them after the closing curly bracket, before the semicolon, like this:

```
struct person {  
    int age;  
    char *name;  
} flavio;
```

Or multiple ones, like this:


```
struct person {  
    int age;  
    char *name;  
} flavio, people[20];
```

In this case I declare a single `person` variable named `flavio` , and an array of 20 `person` named `people` .

We can also declare variables later on, using this syntax:

```
struct person {  
    int age;  
    char *name;  
};  
  
struct person flavio;
```

We can initialize a structure at declaration time:

```
struct person {  
    int age;  
    char *name;  
};  
  
struct person flavio = { 37, "Flavio" };
```

and once we have a structure defined, we can access the values in it using a dot:

```
struct person {  
    int age;  
    char *name;  
};  
  
struct person flavio = { 37, "Flavio" };  
printf("%s, age %u", flavio.name, flavio.age);
```

We can also change the values using the dot syntax:

```
struct person {  
    int age;  
    char *name;  
};  
  
struct person flavio = { 37, "Flavio" };  
  
flavio.age = 38;
```

Structures are very useful because we can pass them around as function parameters, or return values, embedding various variables within them, and each variable has a label.

It's important to note that structures are **passed by copy**, unless of course you pass a pointer to a struct, in which case it's passed by reference.

Using `typedef` we can simplify the code when working with structures.

Let's make an example:

```
typedef struct {  
    int age;  
    char *name;  
} PERSON;
```

The structure we create using `typedef` is usually, by convention, uppercase.

Now we can declare new `PERSON` variables like this:

```
PERSON flavio;
```

and we can initialize them at declaration in this way:

```
PERSON flavio = { 37, "Flavio" };
```

Command line parameters

In your C programs, you might have the need to accept parameters from the command line when the command launches.

For simple needs, all you need to do to do so is change the `main()` function signature from

```
int main(void)
```

to

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

`argc` is an integer number that contains the number of parameters that were provided in the command line.

`argv` is an array of strings.

When the program starts, we are provided the arguments in those 2 parameters.

Note that there's always at least one item in the `argv` array: the name of the program

Let's take the example of the C compiler we use to run our programs, like this:

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

If this was our program, we'd have `argc` being 4 and `argv` being an array containing

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

Let's write a program that prints the arguments it receives:

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
}
```

If the name of our program is `hello` and we run it like this: `./hello`, we'd get this as output:

```
./hello
```

If we pass some random parameters, like this: `./hello a b c` we'd get this output to the terminal:

```
./hello
a
b
c
```

This system works great for simple needs. For more complex needs, there are commonly used packages like **getopt**.

Header files

Simple programs can be put in a single file, but when your program grows larger, it's impossible to keep it all in just one file.

You can move parts of a program to a separate file, then you create a **header file**.

A header file looks like a normal C file, except it ends with `.h` instead of `.c`, and instead of the implementations of your functions and the other parts of a program, it holds the **declarations**.

You already used header files when you first used the `printf()` function, or other I/O function, and you had to type:

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

to use it.

`#include` is a preprocessor directive.

The preprocessor goes and looks up the `stdio.h` file in the standard library, because you used brackets around it. To include your own header files, you'll use quotes, like this:

```
#include "myfile.h"
```

The above will look up `myfile.h` in the current folder.

You can also use a folder structure for libraries:

```
#include "myfolder/myfile.h"
```

Let's make an example. This program calculates the years since a given year:

```
#include <stdio.h>

int calculateAge(int year) {
    const int CURRENT_YEAR = 2020;
    return CURRENT_YEAR - year;
}

int main(void) {
    printf("%u", calculateAge(1983));
}
```

Suppose I want to move the `calculateAge` function to a separate file.

I create a `calculate_age.c` file:

```
int calculateAge(int year) {
    const int CURRENT_YEAR = 2020;
    return CURRENT_YEAR - year;
}
```

And a `calculate_age.h` file where I put the *function prototype*, which is same as the function in the `.c` file, except the body:

```
int calculateAge(int year);
```

Now in the main `.c` file we can go and remove the `calculateAge()` function definition, and we can import `calculate_age.h`, which will make the

`calculateAge()` function available:

```
#include <stdio.h>
#include "calculate_age.h"

int main(void) {
    printf("%u", calculateAge(1983));
}
```

Don't forget that to compile a program composed by multiple files, you need to list them all in the command line, like this:

```
gcc -o main main.c calculate_age.c
```

And with more complex setups, a Makefile is necessary to tell the compiler how to compile the program.

The preprocessor

The preprocessor is a tool that helps us a lot when programming with C. It is part of the C Standard, just like the language, the compiler and the standard library.

It parses our program and makes sure that the compiler gets all the things it needs before going on with the process.

What does it do, in practice?

For example, it looks up all the header files you include with the `#include` directive.

It also looks at every constant you defined using `#define` and substitutes it with its actual value.

That's just the start, and I mentioned those 2 operations because they are the most common ones. The preprocessor can do a lot more.

Did you notice `#include` and `#define` have a `#` at the beginning? That's common to all the preprocessor directives. If a line starts with `#`, that's taken care by the preprocessor.

Conditionals

One of the things we can do is to use conditionals to change how our program will be compiled, depending on the value of an expression.

For example we can check if the `DEBUG` constant is 0:

```
#include <stdio.h>

const int DEBUG = 0;

int main(void) {
    #if DEBUG == 0
        printf("I am NOT debugging\n");
    #else
        printf("I am debugging\n");
    #endif
}
```

Symbolic constants

We can define a **symbolic constant**:

```
#define VALUE 1
#define PI 3.14
#define NAME "Flavio"
```

When we use `NAME` or `PI` or `VALUE` in our program, the preprocessor replaces its name with the value, before executing the program.

Symbolic constants are very useful because we can give names to values without creating variables at compilation time.

Macros

With `#define` we can also define a **macro**. The difference between a macro and a symbolic constant is that a macro can accept an argument and typically contains code, while a symbolic constant is a value:

```
#define POWER(x) ((x) * (x))
```

Notice the parentheses around the arguments, a good practice to avoid issues when the macro is replaced in the precompilation process.

Then we can use it in our code like this:

```
printf("%u\n", POWER(4)); //16
```

The big difference with functions is that macros do not specify the type of their arguments or return values, which might be handy in some cases.

Macros however are limited to one line definitions, and

If defined

We can check if a symbolic constant or a macro is defined using `#ifdef` :

```

#include <stdio.h>
#define VALUE 1

int main(void) {
#ifdef VALUE
    printf("Value is defined\n");
#else
    printf("Value is not defined\n");
#endif
}

```

We also have `#ifndef` to check for the opposite (macro not defined).

We can also use `#if defined` and `#if !defined` to do the same task.

It's common to wrap some block of code into a block like this:

```

#if 0

#endif

```

to temporarily prevent it to run, or to use a DEBUG symbolic constant:

```

#define DEBUG 0

#if DEBUG
    //code only sent to the compiler
    //if DEBUG is not 0
#endif

```

Predefined symbolic constants you can use

The preprocessor also defines a number of symbolic constants you can use, identified by the 2 underscores before and after the name, including:

- `__LINE__` translates to the current line in the source code file
- `__FILE__` translates to the name of the file
- `__DATE__` translates to the compilation date, in the `Mmm gg aaaa` format
- `__TIME__` translates to the compilation time, in the `hh:mm:ss` format