Basic C syntax

Raul P. Pelaez

November 4, 2025

Contents

1 Introduction

Let us go through some of the basic syntax in C, as well as basic programming concepts.

Warning

C $\neq$ C++
Keep in mind that C and C++ are siblings but distinct. Some habits from C++ do not apply in C and vice versa. A few key examples:

- Input/Output (`printf/scanf` in C vs `std::cout/std::cin` in C++)
- Strings (`char[]/char *` in C vs `std::string` in C++)
- Dynamic storage (`malloc/free/realloc/calloc` in C vs `std::vector` and RAII in C++)

Advice

Do not ChatGPT this document
It is essential that you are able to follow and understand this document **unaided**. If you feel like you need an LLM to be able to understand the information in this document, you should consider it a profound skill-issue that you must address. Reading comprehension is a skill that you must develop now. If you do not understand something, re-read it, look for other resources, ask a human. Do not rely on LLMs to do your thinking for you at this stage in your development, lest your skills atrophy.
Fighting through this document yourself might be frustrating and time consuming for you now. Rest assured this is normal and expected, and a necessary part of your growth as a computer scientist. **Do not take shortcuts**.
Beware of thinking: "I could totally understand these documents myself, but I am going to be lazy and make an LLM summarize it for me". Please, do not lie to yourself. Challenge yourself, are you sure you are up for it?.

Info

Tooling and prerequisites
In this document, I am assuming you have access to a developer environment with a C compiler, a text editor, and a terminal. You should be able to compile and run C code in your environment. If this is not the case, refer to the Tooling notes you should have received with this document.

1.1 Trusty online resources

A general word of advice: Be extremely skeptic about online information. Most resources you will find in the wild will be outdated, wrong, or just plain bad. Detecting good resources is a skill that you will develop with time. In a similar note, be very critical of code generated by LLMs, which have been trained in part with all these terrible resources.

Here are some resources that you can trust:

- `https:\\cppreference.com` also has an excellent C section. Look here when you want to know more about a function, header, or macro in the C standard library.
- `https:\\godbolt.org` is a great tool to see how your code is compiled, and even compare different compilers and flags. You can also see the assembly code generated by your code.
- `https:\\www.open-std.org/jtc1/sc22/wg14/` is the C standard (WG14) working group page. The standard drafts live here. Beware, the standard reads like a legal document.

Advice

LLMs are trained on terrible code
LLMs have been trained on, essentially, all the code available in the internet. Anyone can post code to the internet. Look around, judge the quality of the code you find. It is terrible, I know. LLMs have learned from this terrible code. We have applied endless band-aids to the models to try and mitigate for this, but the issue is at the heart of the technology. You must be aware of this bias!

2 The Von Neumann architecture

C is a low-level language that is close to the hardware. It is designed to be efficient and to give the programmer control over the machine. To understand C, it is useful to understand the Von Neumann architecture, which is the basis for most modern computers. Let me give you a very brief overview of it.

The Von Neumann architecture, models memory as a single, linear sequence of addressable cells (a 1D tape) where both data and instructions share the same memory space. A Von Neumman machine will start at a particular "entry" memory address, and will undergo a continuous cycle of fetching, decoding, and executing instructions (in the CPU) sequentially from memory. For that, the machine has a program counter (PC) that points to the current instruction being executed. In principle, the PC is incremented after each instruction, but some instructions specifically modify the PC (like jumps, function calls, returns, etc).

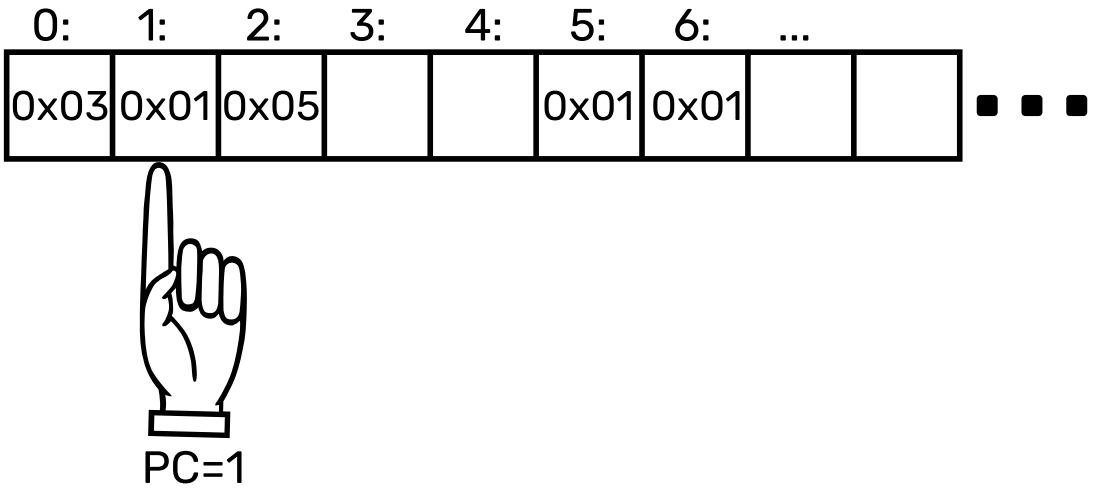

Figure 1: A representation of the memory tape of a Von Neumann machine. The hand represents the program counter (PC) that points to the current instruction being executed. Each cell, identified by its address (above), can hold a byte of information (inside). Some cells might be empty, some might hold data, and some might hold instructions.

A program can be thus be encoded as a sequence of bytes in a memory tape, along with the information of where the PC should start executing. The program is loaded into memory, the PC is set to the entry point, and the machine starts executing instructions sequentially. Imagine that each cell in memory can hold a byte of information (8 bits), which can represent data or instructions[1]. The decoding mechanism will read the bytes at the address pointed by the PC, interpret them as an instruction, and execute it. The decoder has a table, called the instruction set architecture (ISA), that maps byte patterns to instructions. For example, imagine that the byte pattern `0x01` represents the `jump` instruction, which instructs the machine to jump to the address in memory specified by the next byte (you see, memory and data in the same tape). Can you see how the program in figure 1 represents an infinite loop?

This jumping around in memory is the basis of control flow in programming. By using jumps, we can implement loops, conditionals, or call functions and return from them. You see, all that C does, ultimately, is to provide a way to write these sequences of instructions in a human-readable way, and then compile them into the corresponding byte patterns that the machine can understand. Luckily, we do not have to write these byte patterns ourselves, we can write code and let the compiler do the work for us. Still, it is useful to understand how the machine works.

[1] Note that in reality, these cells might hold more than a byte, or instructions might span more than one cell

Insight

This model is not particular to C. Any code we write in any language will be ultimately compiled or interpreted into a sequence of instructions that the machine can understand in a very similar way as in figure 1. The difference between languages is how they allow us to express these instructions, and how much control they give us over the machine. In essence, in how we abstract the machine.

3 About C standards (dialects) and compilation

C is a changing language. Since its initial design in the early 1970s, it has undergone several revisions and updates to improve its features, fix bugs, and adapt to new programming paradigms. C has evolved from C89 to C99, C11, C17 (a bug-fix update to C11), and most recently C23. We will target C17 and opportunistically use C23 features **if** your toolchain supports them.

Your compiler will default to a certain dialect unless explicitly instructed otherwise. In order to change it you must enable the correct flag, which depends on your compiler.

Remember that we compile a source code called "code.c" with:

```
$ gcc code.c -o code # With the GNU compiler
$ clang code.c -o code # Or with Clang/LLVM
$ cc code.c -o code  # cc is often an alias to gcc or clang
# In Windows you might have the MSVC compiler
PS> CL /Fe"path/to/code" code.c
```

In order to enforce a particular C standard, we add the "std" flag (adjust to your compiler's support):

```
$ cc -std=c17 code.c -o code
$ cc  -std=c23 code.c -o code   # or sometimes -std=c2x on older releases
```

You might want to enable warnings: `-Wall -Wextra -Wpedantic`, which will make the compiler output many warnings when it detects potentially dangerous code.

Advice

Many online guides will fail to mention which C standard they are using. The author might be using a very old standard, which will result in you learning outdated practices. If you see no mention whatsoever to the C standard, assume it is C89 or C99, and be very skeptical of the information you are reading.

4 Basic structure of a C program

A C program is composed of functions, declarations and statements. The entry point of a C program is the `main` function, which is where the execution starts. All *statements* in C must end with a semicolon `;`. A minimal C program, that prints "Hello, World!" to the terminal, looks like this:

```c
#include <stdio.h>  // Preprocessor directive to include the standard input/output library
int main() { // Function declaration of main, the entry point of the program
    // Function body
    printf("Hello, World!\n");  // Print "Hello, World!" to the console
    return 0;  // Return 0 to indicate successful execution
}
```

Output

```
Hello, World!
```

Write this code into a file called `hello.c`, compile it in the terminal with `cc hello.c -o hello` and run it with `./hello`.

Warning

Basic computer literacy
I am assuming here that you are familiar enough with your computer as to be able to understand the instructions above. If you are not, please refer to the Tooling notes you should have received with this document.

Info

The `.` command
When writing `./hello`, the `.` means "the current directory". It is a shorthand to tell the terminal to look for the executable in the current directory. If you just write `hello`, the terminal will look for the executable in the directories listed in the `PATH` environment variable, which usually does not include the current directory for security reasons. When the terminal receives `./hello`, it looks for the `hello` executable in the current directory and runs it.

Info

Comments
Comments in C start with `//` for single-line comments, or `/* ... */` for multi-line comments. The compiler ignores comments, they are just for humans to read.
Example

```
// This is a single-line comment
/*
    This is a multi-line comment
    It can span multiple lines
*/
```

There is a lot to unpack in this simple program. Let us go through it line by line.

4.1 Including headers

Info

Preprocessor directives
The first line, starting with the `#` symbol, is known as a preprocessor directive. The preprocessor is a tool that runs as part of the compilation process, before the actual compilation. It processes these directives and modifies the source code accordingly.

The `#include <stdio.h>` directive tells the preprocessor to include the contents of the `stdio.h` header file (standard input/output), which contains declarations for standard input/output functions like `printf` (formatted print) and `scanf` (formatted *scan*, as in taking input from the terminal). Includes are the way in C to *import* code from other files. We usually call these "importable" files "headers", and they usually have the `.h` extension. Some headers are available to us always, like `stdio.h`, which is part of the C standard library. Other headers might be provided by third-party libraries, or you might write your own headers to organize your code.

Info

Angle brackets vs quotes in includes
When including a header, you can use either angle brackets `<...>` or double quotes `"..."`. The difference is that angle brackets tell the preprocessor to look for the header in the system directories, while double quotes tell it to look in the current directory first, and then in the system directories. Use angle brackets for standard library headers and third-party library headers, and use double quotes for your own headers.

4.2 The `main` function

The next line, `int main(void) {`, is the declaration of the `main` function. Every C program must have a `main` function, as it is the entry point of the program. Roughly speaking, this will be the place where the hand in figure 1 starts executing instructions.

Do not worry too much about what a function *is* or the specific syntax of this line. The important thing is that the body of the `main` function, denoted with a *block* (the curly braces `{...}`), contains the code that will be executed when the program runs. The code inside the block is called the *body* of the function. When the `return` instruction is issued, the `main` function sends the execution "back to where it was called from", which in the case of the `main` function means exiting the program. The `0` is a convention to indicate that the program finished successfully. Other values can be used to indicate different types of errors.

In this particular case, the main function is going to *call* a function called `printf`, which is declared in the `stdio.h` header we included before. The `printf` function takes a *string* as an argument (the text to print) and prints it to the terminal. The string we are passing to `printf` is `"Hello, World!\n"`, which is a sequence of characters enclosed in double quotes. The `\n` at the end of the string is a special character called a *newline*, which tells the terminal to move to the next line after printing the string. Think about figure 1, the hand, when encountering the instruction `printf`, is going to jump somewhere else in memory, where the code for the `printf` function is located, execute that code, and then return to the next instruction after the `printf` call.

We have talked about functions, forced by the essential nature of the `main` function in the C language. But what *is* a function?

5 Functions

Imagine that there is a task that you need to perform constantly in a program you are building. See for instance the following snippet:

```c
#include <stdio.h>
int main() {
    printf("Hello, Alice!\n");
    printf("Hello, Bob!\n");
    printf("Hello, Charlie!\n");
    return 0;
}
```

Output

```
Hello, Alice!
Hello, Bob!
Hello, Charlie!
```

Is it not tedious to have to write "Hello" every time? and what if we want to be more informal and change it to "Hi"? We would have to change it in every line. This is where functions come in handy. A function is a named block of code that performs a specific task and can be reused multiple times. We can define a function that takes a name as an argument and prints "Hello, <name>!" to the terminal. Here is how we can do it:

```c
#include <stdio.h>
void greet(const char *name) { // Function declaration of greet
    printf("Hello, %s!\n", name); // Print "Hello, <name>!" to the console
}

int main() {
    greet("Alice"); // Call the greet function with "Alice" as argument
    greet("Bob");   // Call the greet function with "Bob" as argument
    greet("Charlie"); // Call the greet function with "Charlie" as argument
    return 0;
}
```

Output

```
Hello, Alice!
Hello, Bob!
Hello, Charlie!
```

Much better! Now we can change the greeting in one place, and it will be reflected everywhere. We can also reuse the `greet` function in other parts of our program, or even in other programs. Keep figure 1 in mind, the `greet` function is just a block of instructions with a known address located *somewhere* in the tape. Each time we *call* the function, the hand jumps to that address, executes the instructions there, and then returns to the next instruction after the call.

You probably realized it is a bit more complicated than that. It is not exactly the same instructions that are being executed, since each time something different is being printed. When the function is called, the hand is somehow jumping while carrying some context, in this case the name to be printed. This context is passed to the function as an *argument*. In our case, *something* called `name` is allowed as context for the `greet` function. When writing `greet("Alice")`, we are calling the `greet` function with the argument "Alice". Inside the function, the argument is accessible through the parameter `name`, which is declared in the function definition.

Info

Function calls

A function is called by writing its name followed by parentheses `()`. If the function takes parameters, we pass the arguments inside the parentheses, separated by commas. For example, to call the `greet` function with the argument "Alice", we write:

```
greet("Alice");
```

Info

Function definitions

The syntax for defining a function is:

```
return_type function_name(parameter_list) {
    // Function body
    return value; // Optional, if the return type is not void
}
```

Where:

- `return_type` is the type of value that the function returns. If the function does not return a value, we use `void`.
- `function_name` is the name of the function.
- `parameter_list` is a comma-separated list of parameters that the function takes as input. Each parameter has a type and a name.
- The function body is a block of code enclosed in curly braces `{...}` that contains the instructions to be executed when the function is called.
- The `return` statement is used to return a value from the function. It is optional if the return type is `void`. The caller of the function can use the returned value.

Info

Function declaration

We can state the existence of a function without defining it, this is called a function declaration. It is useful when we want to use a function that is defined in another file or later in the same file. A function declaration has the same syntax as a function definition, but without the body. For example, we can declare the `greet` function before the `main` function like this:

```
void greet(const char *name); // Function declaration of greet
```

Info

Functions can only return one value
In C, a function can only return one value. If we want to return multiple values, we can use pointers or structures. These concepts are beyond the scope of this document, but we will touch on them in future lessons.

6 Variables and types

C is a statically typed language, which means that every variable must have a type, and the type must be known at compile time. A variable is a named location in memory that can hold a value of a specific type. The type of a variable determines how much memory it occupies, how it is represented in memory, and what operations can be performed on it.

The basic types in C are:

- `int`: integer type, used to represent whole numbers. It usually occupies 4 bytes (32 bits) in memory, but this can vary depending on the architecture.
- `float`: floating-point type, used to represent real numbers with decimal points. It usually occupies 4 bytes (32 bits) in memory.
- `double`: double-precision floating-point type, used to represent real numbers with more precision than `float`. It usually occupies 8 bytes (64 bits) in memory.
- `char`: character type, used to represent single characters. It usually occupies 1 byte (8 bits) in memory. Characters are represented using the ASCII or Unicode encoding.
- `void`: special type that represents the absence of a value. It is used in functions that do not return a value.
- `const char*`: string type, used to represent strings of characters. Although the syntax is funny, at this point you can read this as a special type that represents strings.

You can declare a variable by specifying its type followed by its name. For example:

```c
#include <stdio.h>
int main() {
    int age = 25; // Declare an integer variable called age and initialize it to 25
    float height = 1.75; // Declare a float variable called height and initialize it to 1.75
    char initial = 'R'; // Declare a char variable called initial and initialize it to 'R'
    const char *name = "Raul"; // Declare a string variable called name and initialize it to "Raul"
    printf("Name: %s\n", name);
    printf("Age: %d\n", age);
    printf("Height: %.2f\n", height);
    printf("Initial: %c\n", initial);
    return 0;
}
```

Output

```
Name: Raul
Age: 25
Height: 1.75
Initial: R
```

Info

Format specifiers
In the `printf` function, we use format specifiers to indicate how to format the output. The format specifiers are preceded by the `%` symbol and are replaced by the corresponding argument in the function call. The most common format specifiers are:

- `%d`: integer
- `%f`: float
- `%.2f`: float with 2 decimal places
- `%c`: char
- `%s`: string

For example, in the line `printf("Age:  %d\n", age);`, the `%d` is replaced by the value of the `age` variable, which is 25.

Info

More on assigning values to variables
You can assign a value to a variable when you declare it, as we did in the examples above. This is called initialization. You can also assign a value to a variable after it has been declared, using the assignment operator `=`. For example:

```c
#include <stdio.h>
int main() {
    int age; // Declare an integer variable called age
    age = 25; // Assign the value 25 to the age variable
    printf("Age: %d\n", age);
    age = age + 1; // Increment the age variable by 1
    printf("Next year, age: %d\n", age);
    return 0;
}
```

Output

```
Age: 25
Next year, age: 26
```

Note the last assignation, which is usually confusing for beginners. The right-hand side of the assignment is evaluated first, and then the result is assigned to the variable on the left-hand side. In this case, we are taking the current value of `age`, adding 1 to it, and then assigning the result back to `age`. This is a common pattern in programming, and it is often used to increment or update the value of a variable. Mathematically, the statement `age = age + 1;` does not make sense, but in programming it is perfectly valid. The `=` operator is not an equality operator (as in math), it is an assignment operator. It is used to assign a value to a variable, not to compare two values or provide a statement of truth.

7 Coming back to functions

Now that we have learned about types, you might have a better time understanding the following code:

```c
#include <stdio.h>

//Function declaration
float rectangle_area(float width, float height);

int main() {
  // Declare a float variable called w and initialize it to 5.0
  float w = 5.0;
  // Declare a float variable called h and initialize it to 3.0
  float h = 3.0;
  // Call the rectangle_area function with w and h as arguments
```

```c
  float area = rectangle_area(w, h);
  printf("Width: %.2f, Height: %.2f, Area: %.2f\n",
         w, h, area);
  return 0;
}

// Function definition
float rectangle_area(float width, float height) {
    float result = width * height; // Calculate the area
    return result; // Return the area
}
```

Output

```
Width: 5.00, Height: 3.00, Area: 15.00
```

Some new stuff in this code. Lets go through it: We are *declaring* a function called `rectangle_area` that takes two `float` parameters, with names `width` and `height`, and returns a `float` value. The function calculates the area of a rectangle by multiplying the width and height.

In the `main` function, we declare two `float` variables, `width` and `height`, and initialize them to 5.0 and 3.0 respectively. We then call the `rectangle_area` function with these variables as arguments, and store the returned value in a new variable called `area`. Finally, we print the width, height, and area to the terminal using the `printf` function.

Advanced

Printf is a variadic function
The `printf` function is a special type of function called a variadic function, which means that it can take a variable number of arguments. The first argument is always the format string, which contains the text to be printed and the format specifiers. The remaining arguments are the values to be printed, which are matched to the format specifiers in order. The syntax to define our own variadic functions is a bit more complicated, and we will not cover it in this document.

Note that we decided (which is not mandatory) to separate the function declaration from the function definition. The declaration tells the compiler that the function exists and what its signature is (name, parameters, return type), while the definition provides the actual implementation of the function. This separation is useful when we want to organize our code in multiple files, or when we want to declare functions before they are defined. We also arbitrarily decided to *define* the function after the `main` function. This is also not mandatory, we could have defined it before the `main` function, or even in another file.

The function is multiplying the two parameters and storing the result in a new variable called `result`, which is then returned to the caller. The caller can use the returned value as needed. In this case, we store it in the `area` variable and print it to the terminal.

Note how there is a disconnection between the *name* of the parameters in the function declaration and definition, and the *name* of the arguments in the function call. Let me explain to you with an allegory why this is not an issue, and it is actually what you want.

> Imagine that you (a program) are a barista in a coffee shop. This coffee shop is special, because customers must bring their own milk. You can prepare a coffee for a customer (a function), with any milk they bring (a parameter).
>
> You have a recipe (the function definition) that tells you how to prepare the coffee with a given milk. When a customer arrives, they will hand to you a container with milk (an argument), say almond milk. You take the almond milk, follow the recipe, and prepare the coffee. After following the steps in the recipe, you hand the coffee (the return value) back to the customer (the caller).
>
> The customer does not care about how you refer to the milk (the parameter name), they just care about the coffee they receive. You do not care what the customer calls their milk (the

name of the variable holding the argument), you just care about following the recipe with the milk they provide.

Insight

You might find I am going too slow with this concept, but in my experience, this is one of the most difficult concepts for beginners to grasp.

Let me also give you another example, to understand why the names of parameters and arguments do not have to match:

```c
#include <stdio.h>

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

int main() {
    int sum = add(1, 5);
    printf("Sum: %d\n", sum);
    return 0;
}
```

Output

```
Sum: 6
```

This code runs without issue and does what we expect. Note that we did not even had to *assign* the arguments to named variables before calling the function. We just passed the values directly. The parameter names `a` and `b` are just the way the function refers to the provided arguments internally.

8 Separating code in multiple files

As programs grow in size, it is useful to separate code in multiple files. This helps to organize the code, and to avoid having to recompile everything when only a small part of the code changes. In C, we usually separate code in two types of files: source files and header files.

- Source files have the `.c` extension, and contain the implementation of functions and variables.
- Header files have the `.h` extension, and contain the declarations of functions and variables.

Lets separate the previous example in three files: `main.c`, `add.c`, and `add.h`.

`add.h`

```c
#ifndef ADD_H // Include guard to prevent multiple inclusion
#define ADD_H

/**
 * Adds two integers and returns the result.
 *
 * @param a The first integer.
 * @param b The second integer.
 * @return The sum of a and b.
 */
int add(int a, int b);
#endif // ADD_H
```

Info

Include guards
The include directive will just copy-paste the contents of the included file. If a header is included multiple times, it will result in multiple definitions of the same function or variable, which will cause a compilation error. To prevent this, we use include guards, which are preprocessor directives that check if a unique identifier (usually the name of the header in uppercase) is defined. If it is not defined, it defines it and includes the contents of the header. If it is already defined, it skips the contents of the header. This way, the header is included only once, even if it is included multiple times in different files.

Info

Documentation comments
The comment block before the function declaration is a documentation comment, which is a special type of comment that describes the purpose and usage of the function. It is not mandatory, but it is a good practice to document your code, especially for public APIs. There are tools that can generate documentation from these comments, like Doxygen.
VSCode also makes use of these. If you hover your cursor over an usage of `add` as defined in the example above, you will get a rendered version of the Doxygen comment. Looks pretty professional, right?

`add.c`

```c
#include "add.h" // Include the header file
int add(int a, int b) { // Function definition
    return a + b; // Return the sum of a and b
}
```

`main.c`

```c
#include <stdio.h>
#include "add.h" // Include the header file
int main() {
    int sum = add(1, 5); // Call the add function
    printf("Sum: %d\n", sum); // Print the result
    return 0;
}
```

Note that, as long as `main.c` is concerned, the `add` function is not defined. This is not a problem because we have included the `add.h` header, which declares the function. The compiler will check that the function is called with the correct number and types of arguments, and that the return type is compatible with the variable it is assigned to. The actual implementation of the function is in `add.c`, which will be compiled separately. When compiling, we must compile both source files separately and link them together to create the final executable. We can do this like so:

```
# Compile (-c) both source files into object files
$ cc -c main.c -o main.o
$ cc -c add.c -o add.o
# Link the object files into a single executable
$ cc main.o add.o -o program
# Run the program
$ ./program
```

9 Conditionals: controlling the flow of execution

Conditionals are used to control the flow of execution in a program based on certain conditions. The most common conditional statements in C are:

- `if`: executes a block of code if a condition is true
- `else`: executes a block of code if the condition in the preceding `if` statement is false

- `else if`: executes a block of code if a different condition is true
- `switch`: executes one of many blocks of code based on the value of a variable
  ** If, else, else if

The syntax for an `if` statement is:

```
if (condition) {
    // Code to be executed if the condition is true
} else if (another_condition) {
    // Code to be executed if the another_condition is true
} else {
    // Code to be executed if none of the above conditions are true
}
```

The `condition` is an expression that evaluates to either true (non-zero) or false (zero). If the condition is true, the code inside the block is executed. If the condition is false, the program checks the next `else if` condition, and so on. If none of the conditions are true, the code inside the `else` block is executed.

C provides special operators to compare values and combine conditions:

- `==`: equality operator, checks if two values are equal
- `!=`: inequality operator, checks if two values are not equal
- `<`: less than operator, checks if the left value is less than the right value
- `>`: greater than operator, checks if the left value is greater than the right value
- `<=`: less than or equal to operator, checks if the left value is less than or equal to the right value
- `>=`: greater than or equal to operator, checks if the left value is greater than or equal to the right value
- `&&`: logical AND operator, checks if both conditions are true
- `||`: logical OR operator, checks if at least one condition is true
- `!`: logical NOT operator, negates a condition

All of the above operators yield a boolean result (1 for true, 0 for false). All of them are binary operators, except for the logical NOT operator, which is unary (only takes one operand, like so: `!condition`). For instance:

```
#include <stdio.h>
int main() {
    int age = 20;
    if(!(age>0)){ // Check if age is not greater than 0
        printf("Age must be a positive number.\n");
        return 1; // Exit the program with an error code
    }
    if (age < 18) {
        printf("You are a minor.\n");
    } else if (age >= 18 && age < 65) {
        printf("You are an adult.\n");
    } else {
        printf("You are a senior citizen.\n");
    }
    return 0;
}
```

Output

```
You are an adult.
```

Try to take this code to Godbolt and play around with the value of age. See how the output changes based on the value of age.

9.1 Switch

The `switch` statement is used to execute one of many blocks of code based on the value of a variable. The syntax for a `switch` statement is:

```
switch (variable) {
    case value1:
        // Code to be executed if variable == value1
        break;
    case value2:
        // Code to be executed if variable == value2
        break;
    // More cases...
    default:
        // Code to be executed if variable does not match any case
}
```

The `variable` is an expression that evaluates to a value. The `case` statements define the possible values that the variable can take, and the code inside the block is executed if the variable matches the value. The `break` statement is used to exit the `switch` statement after executing a case. If none of the cases match, the code inside the `default` block is executed.

Here is an example:

```
#include <stdio.h>
int main() {
    int day = 3;
    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        // All other days...
        default:
            printf("Another day\n");
    }
    return 0;
}
```

Output

```
Wednesday
```

You can think of a switch statement as a more efficient way to write multiple `if` statements when checking the value of a single variable. Its also more readable than a long chain of `if` and `else if` statements in some cases.

10 Loops

Loops are used to execute a block of code multiple times based on a condition. Loops are useful when we want to repeat a task until a certain condition is met, or when we want to iterate over a collection of items. Beginners often struggle understanding the concept of loop, so read this section carefully.

The most common loop statements in C are:

- `for`: executes a block of code a specific number of times
- `while`: executes a block of code while a condition is true
- `do...while`: executes a block of code at least once, and then while a condition is true

10.1 For loop

The syntax for a `for` loop is:

```c
for (initialization; condition; increment) {
    // Code to be executed in each iteration
}
```

The `initialization` is executed once at the beginning of the loop, and is usually used to declare and initialize a loop variable. The `condition` is checked before each iteration, and if it is true, the code inside the block is executed. The `increment` is executed after each iteration, and is usually used to update the loop variable. Here is an example:

```c
#include <stdio.h>
int main() {
    for (int i = 0; i < 5; i++) {
        printf("Iteration %d\n", i);
    }
    return 0;
}
```

Output

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Info

The ++ operator
In C, the `++` operator is a shorthand way to increment a variable by 1. Writting `i++` is equivalent to writting `i = i + 1` or `i += 1`.

The execution here goes like this:

1. The loop variable `i` is defined and initialized to 0.
2. The body of the loop is executed with `i` having the value 0, printing "Iteration 0".
3. The increment statement `i++` is executed, increasing the value of `i` to 1.
4. The condition `i < 5` is checked, and since it is true, the body of the loop is executed again, but this time with `i` having the value 1, printing "Iteration 1".
5. Steps 3 and 4 are repeated until the condition `i < 5` is false (when `i` becomes 5). At this point, the loop ends and the program continues with the next statement after the loop.

Advanced

Controlling loop execution

We have two special statements to control the execution of loops:

- `break`: exits the loop immediately, regardless of the condition
- `continue`: skips the current iteration and goes to the next iteration

Here is an example that uses both:

```c
#include <stdio.h>
int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break; // Exit the loop when i is 5
        }
        if (i % 2 == 0) {
            continue; // Skip even numbers
        }
        printf("Odd number: %d\n", i);
    }
    return 0;
}
```

Output

```
Odd number: 1
Odd number: 3
```

Note that we could had simply written `for (int i = 1; i < 5; i+=2)` to iterate only over odd numbers smaller than 5, but this is just an example to illustrate the usage of `break` and `continue`.

10.2 While loop

The syntax for a `while` loop is:

```c
while (condition) {
    // Code to be executed in each iteration
}
```

The `condition` is checked before each iteration, and if it is true, the code inside the block is executed. The loop continues until the condition is false. Here is an example:

```c
#include <stdio.h>
int main() {
    int i = 0; // Initialize the loop variable
    while (i < 4) { // Check the condition
        printf("Iteration %d\n", i); // Execute the loop body
        i++; // Increment the loop variable
    } // Go back to the condition until it is false
    return 0;
}
```

Output

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
```

10.3 Do...while loop

The syntax for a `do...while` loop is:

```c
do {
```

```
    // Code to be executed in each iteration
} while (condition);
```

The code inside the block is executed at least once, and then the `condition` is checked. If the condition is true, the loop continues and the code inside the block is executed again. This continues until the condition is false. Here is an example:

```c
#include <stdio.h>
int main() {
    int i = 0; // Initialize the loop variable
    do { // Execute the loop body
        printf("Iteration %d\n", i);
        i++; // Increment the loop variable
    } while (i < 4); // Check the condition after the loop body
    return 0;
}
```

Output

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
```

This structure is useful when we want to ensure that the code inside the block is executed at least once, regardless of the condition.

11 Scope

Scope refers to the visibility and lifetime of variables and functions in a program. In C, there are two types of scope: global scope and local scope.

- Global scope: variables and functions declared outside of any function have global scope, which means they can be accessed from any function in the same file or in other files (if declared with the `extern` keyword).
- Local scope: variables and functions declared inside a block have local scope, which means they can only be accessed from within that function.

Info

Block
A block is a section of code enclosed in curly braces `{ }`. Blocks are used to group statements together, and they define a new scope for variables and functions declared inside them. For example, the body of a function is a block, and the body of an `if` statement or a loop is also a block.

Here is an example that illustrates the concept of scope:

```c
#include <stdio.h>
int global_var = 10; // Global variable
void print_global() {
    int variable_inside_function = 20; // Local variable
    printf("Global variable: %d\n", global_var); // Access global variable
}
int main() {
    int local_var = 5; // Local variable
    print_global(); // Call function to print global variable
    printf("Local variable: %d\n", local_var); // Access local variable
    // Error: variable_inside_function is not accessible here
    // printf("Variable inside function: %d\n", variable_inside_function);
    return 0;
}
```

Output

```
Global variable: 10
Local variable: 5
```

Inspect this code closely. In general, the message is that a variable can only be seen from the block it is defined in, and any blocks defined inside it. For instance, `global_var` is defined in the global scope, so it can be accessed from any function in the same file. `local_var` is defined inside the `main` function, so it can only be accessed from within that function. `variable_inside_function` is defined inside the `print_global` function, so it can only be accessed from within that function.

Advice

Avoid global variables
Global variables can make code harder to understand and maintain, as they can be modified from anywhere in the program. It is generally a good practice to minimize the use of global variables, and to prefer passing variables as parameters to functions or using local variables instead.
It is tempting when starting to code to think of global variables as the solution to many problems (such as sharing data between functions). However, this is a bad habit that will lead to unmaintainable code. Avoid it at all costs. When you find yourself thinking "I will solve this with a global variable" it is usually a sign that your current approach/design is flawed and you should rethink it.

12 Problem solving 101

Perhaps the main bottleneck for a beginner is the disconnection between the theoretical knowledge of programming concepts/syntax and its application to solving a real problem. For instance, you might be asked the following:

> Write a program that sorts the numbers of a list of integers input by the user in the command line. The program should end by printing the sorted list.

Self assess here. Are you able to express this into code? The actual programming knowledge you need to accomplish this is not very profound. You probably know enough syntax to get it done. However, it is probable that you are thinking: "I do not know where to even start". This is normal. Mastery of this process touches many intellectual fields of knowledge, such as software architecture, software design, critical thinking, software engineering... Many of which could be considered "soft" skills that are typically not covered in programming courses and have a heavy experience component.

Let me introduce you to a basic intellectual strategy to tackle any problem.

12.1 Divide and conquer

The human mind is simply not prepared to solve complex problems. We are very good, though, in solving *simple* problems. You can hack your brain using this information by simply splitting your problems into smaller, more manageable ones.

You do this constantly without even realizing. For instance, going to the university for a lecture is an extremely complex task, which you split without effort. Let me make it explicit. In order to get to the university (a super complex problem), you must:

- Get out of the house
- Take the metro
- Enter the university
- Go to the assigned classroom

These steps are *simpler* than the final goal, but still we can split them into simpler ones. For instance, in order to get out of the house, you may have to:

- Take a shower
- Get dressed

- Have breakfast
- Brush your teeth
- Cross the front door and lock it

You can go even deeper. In order to "Cross the front door and lock it" you must:

- Walk towards the door
- Grab the handle
- Spin it
- Open the door
- etc

To "Grab the handle" you need to activate the muscles in your hand, and so on and so forth. This hierarchical decomposition of problems is something you do naturally, constantly and largely unconsciously. You can employ this strategy to solve programming problems as well.

Advice

LLMs are good with simple problems
LLMs are the same way, they will typically fail miserably if you ask them to solve anything moderately complex. They will do a perfect job, however, if you request a simple task like "write a function that sums two numbers" or "add doxygen comments to this function".
Learning to understand your problems as a hierarchy of simple ones will also result in a much more effective usage of your code-helpers (AI-based or not).

Insight

In Computer Science, there is an algorithm design strategy called Divide-and-Conquer. The point I want to make is a generalization of this concept to *any* other problem you face.

To apply this to our sorting problem, we can split it into smaller problems:

- Start a development environment
- Read the list of integers from the command line
- Sort the array of integers
- Print the sorted array

Each of these problems can be further split into smaller ones. For instance, to "Read the list of integers from the command line", we can:

- Get the number of arguments passed to the program
- Allocate memory for an array of integers
- Loop through the arguments and convert them to integers
- Store the integers in the array

You can keep splitting the problems until you reach a level where you feel comfortable solving them. My proposal above is just an example, you might find other ways to split the problems that work better for you. This last sentence is key. If you try to "get help" online or via an LLM (or even via a senior) on the whole problem, you will get strategies that are not aligned to your current knowledge, skills, and world model (how **you** understand the world personally). Thus you will have a harder time making practical use of the information you receive.

In the example above, you may not know how to "Get the number of arguments passed to the program", but this is a much more simple search than "How to write a program that X and Y and Z in C". An LLM will also give you a pretty good rundown of this small and specific step.

To start a development environment, you need what I usually refer to in class as "the usual workflow", which consists in:

- Opening the terminal
- Navigating/creating a folder for the project
- Activating/creating a conda environment
- Opening VSCode in the project folder

Again, its much more sensible to ask (ChatGPT or else) "How to open a terminal in Windows/Mac/Linux" than "How to open a development environment for C programming". The latter is too complex, vague and you will get lost in the details.

I know this is a lot to take in, but this is the most important skill you can learn as a programmer. The actual programming knowledge (syntax, specific tools...) is secondary. If you master this skill, you will be able to learn any programming language or framework in a matter of days. You will become much more adaptable to things like adopting workflows you are unfamiliar with, or solving problems outside your field of expertise. If you do not, you will struggle for years.

13 Interview-style Exercises

The following exercises are the kind of question you will get during technical interviews for software-related jobs. Typically, you will be required to write these live, in front of the interviewer, and completely unaided (not even intellisense, sometimes not even code highlighting). I have seen sites like godbolt or hackerrank being used for these. The challenges are designed to evaluate your knowledge of the syntax of a certain language, as well as your problem solving and critical-thinking abilities. You are given around 20 mins to solve each one.

When you arrive to a solution, the interviewer typically critics it and often imposes some new requirement that forces you to modify the implementation.

Insight

Instead of 1-hour technical interviews, some companies will give you take-home challenges instead, which will take you between 2 and 8 hours to complete. In these, you are naturally not supervised and thus can use every tool and aid in your book. However, these are typically followed by a technical interview in which you have to explain your design choices, details of the code, etc, etc. Typically, you get a website with a set of requirements and you are asked to submit a link to a git repo with your implementation. Other times you submit to some internal tool that will run some unit tests on your implementation and give you a report like "your implementation is not passing our tests" and you are given 2 or 3 tries. The challenges are often pretty complex in scope and require around a 1000 lines of code in several languages to complete.
No one likes these challenges, IMHO they are a cautionary tale about how the company will value you if they hire you. These are more about "are you willing to put up with our BS" than "are you skilled". Nonetheless, some companies use these as a first screening and thus you must be ready for them.
Furthermore, these kind of challenges usually involve some obscure tech stack you have never heard of, or strange data structures and requirements. One I came across recently required to write a Python web backend API involving a specific query language and a specific web framework. You might be unfamiliar with the stack, but you are expected to be able to research it, read the docs, etc, in order to solve the task.

Advice

Pair up with a peer. Solve the problems independently and then exchange and critic your implementations. Try to find weak spots, corner cases, etc.
Afterwards, impose some limitations for your peer (e.g. "your solution is inefficient if the input is very large, address that case" or "now the list is made of complex numbers").

Goal

Write a function that returns the second largest element in a sequence.

Goal

Write a function that computes the histogram of a sequence.

Milestone

- Write a parallel version of it.

Goal

Write a program that will generate a vector with random numbers and will compute the sum of all the elements.

Milestone

- Write another version of the function that does not use a loop.

  hint

  Recursion