

INTRODUCTION TO



PROGRAMMING
LANGUAGE



Beta

@c12mentor



Agenda

Intro to programming languages

In depth understanding of high and low level languages

Exploring the fun with C

Compiling C programs

**Once upon a time
programming
languages were
born...**



What is a programming language?

Programming language is a formal language designed to communicate instructions to a computer.

Programming languages can be used to create programs that control the behavior of a machine and to express algorithms precisely.

- Programming languages **define** and **compile** a set of instructions for the CPU for performing any specific task.
- Every programming language has a set of **keywords** along with **syntax**—that it uses for creating **instructions**.

Talking to the computer with instructions....is so fun

- Till now, thousands of programming languages have come into form, with specific **purposes**, and **variations** in terms of the **level of abstraction** that they all provide from the hardware.
- A few of these languages provide **less** or **no abstraction** at all, while the others provide a **very high** abstraction.
- On the basis of this **level of abstraction**, there are two types of programming languages:
 - *Low-level language*
 - *High-level language*

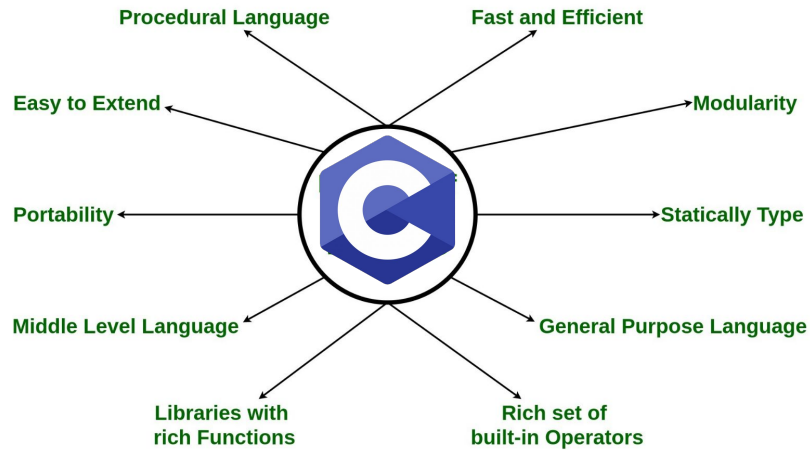
Low and High level programming languages???

- The primary difference between low and high-level languages is that:
 - any programmer (like you) can understand, compile, and interpret a high-level language feasibly as compared to the computers.
 - the computers are capable of understanding the low-level language more feasibly compared to human beings.
- Low level languages: machine code, assembly language,
- High level languages: Java, C++, Python, JavaScript, Ruby, HTML, ...

Difference Between High-Level and Low-Level Languages

Parameter	High-Level Language	Low-Level Language
Basic	These are programmer-friendly languages that are manageable, easy to understand, debug, and widely used in today's times.	These are machine-friendly languages that are very difficult to understand by human beings but easy to interpret by machines.
Ease of Execution	These are very easy to execute.	These are very difficult to execute.
Process of Translation	High-level languages require the use of a compiler or an interpreter for their translation into the machine code.	Low-level language requires an assembler for directly translating instructions of the machine language.
Efficiency of Memory	These languages have a very low memory efficiency. It means that they consume more memory than any low-level language.	These languages have a very high memory efficiency. It means that they consume less energy as compared to any high-level language.
Portability	These are portable from any one device to another.	A user cannot port these from one device to another.
Comprehensibility	High-level languages are human-friendly. They are, thus, very easy to understand and learn by any	Low-level languages are machine friendly. They are, thus, very difficult to understand and learn by any

Features of C Programming Language



C (pronounced /'si:/ – like the letter c)

- C is a high-level, general-purpose programming language created by **Dennis Ritchie** at the **Bell Laboratories** in 1972.
- It is a **very popular language**, despite being old.
- C is strongly **associated with UNIX**, as it was developed to write the UNIX operating system.
- It provides a **straightforward, consistent, powerful interface** for programming systems.
- That's why the C language is widely used for developing system software, application software, and embedded systems.

Applications of C

- It is widely used for developing desktop applications and for developing browsers and their extensions.
- It is widely used in embedded systems.
- It is used to develop databases. MySQL is the most popular database software which is built using C.
- It is used in developing an operating system such as Apple's OS X, Microsoft's Windows
- It is widely used in IoT applications.
- etc

Is C a high OR low-level language?

- The C programming languages come under the category of **middle-level languages**.
- Low-level languages provide little or no abstraction of programming concepts, whereas C programming languages provide the **least degree of abstraction** to **performance** and **efficiency**.
- These abstractions like **macros**, **lambda functions**, **classes** help programmers to use complex functionality in programming without the need for writing more complex code.
- Because of this reason, C (and C++) languages are considered lower-level languages where maximum performance is paramount; however, abstractions are necessary to keep code maintainable and highly readable.

Your first C program

- C is a **compiled programming language**, like Go, Java, Swift or Rust.
- 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.




cont...

alx

```
#include <stdio.h>

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

- we first import the **stdio** library: gives access to input/output functions.
 - **Why have libraries?** 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 e.g stdio and others.
 - **stdio** is the library that provides the **printf()** function.
 - This function is wrapped into a **main()** function. The **main()** function is the **entry point** of any C program.
- 




cont...

alx

```
#include <stdio.h>

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

- **What is a function?** 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. 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, ...);`
- 

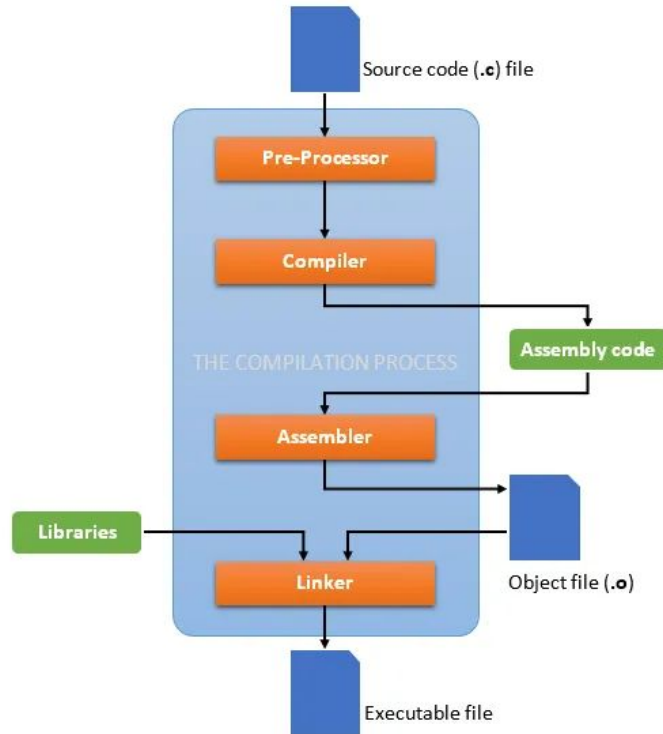
C program execution

```
#include <stdio.h>

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

- The C prog. will be run by the operating system when the program is executed
- To run the program we must first **compile** it. Any Linux or macOS computer already comes with a C compiler built-in.
- A compiler is a special program that translates a programming language's source code into machine code, bytecode or another programming language.
- GCC is the most common C language compiler

C program compilation



Compilation is the translation of source code (the code we write) into object code (sequence of statements in machine language) by a compiler.

The compilation process has four different steps:

1. The preprocessing stage
2. The compiling stage
3. The assembling stage
4. The linking stage

Step 1: Preprocessing -E

- The preprocessor has several roles:
 - it gets rid of all the comments in the source file(s)
 - it includes the code of the header file(s), which is a file with extension `.h` which contains C function declarations and macro definitions
 - it replaces all of the macros by their values
- The output of this step will be stored in a file with a `.i` extension

Step 2: Compiling -S

- The compiler will take the preprocessed file and generate **IR** code (Intermediate Representation), so this will produce a **.s** file.
- Other compilers might produce assembly code at this step of compilation.

Step 3: Assembling -c

- The assembler takes the IR code and transforms it into object code, that is code in machine language (i.e. binary). This will produce a file ending in .o

Step 4: Linking

The linker creates the final executable, in binary, and can play two roles:

- linking all the source files together, that is all the other object codes in the project. For example, if I want to compile `main.c` with another file called `secondary.c` and make them into one single program, this is the step where the object code of `secondary.c` (that is `secondary.o`) will be linked to the `main.c` object code (`main.o`).
- linking function calls with their definitions. The linker knows where to look for the function definitions in the `static libraries` or `dynamic libraries`.

cont..

- By default, after this fourth and last step, that is when you type the whole `gcc main.c` command without any options, the compiler will create an executable program called `a.out`, that we can run by typing `./a.out` in the command line.
- We can also choose to create an executable program with the name we want, by adding the `-o` option to the gcc command, placed after the name of the file or files we are compiling

Resources

1. [High and Low level languages](#)
2. [C – beginner's Handbook](#)
3. [Compiling C files with GCC](#)

**See you at
the next
session!**

