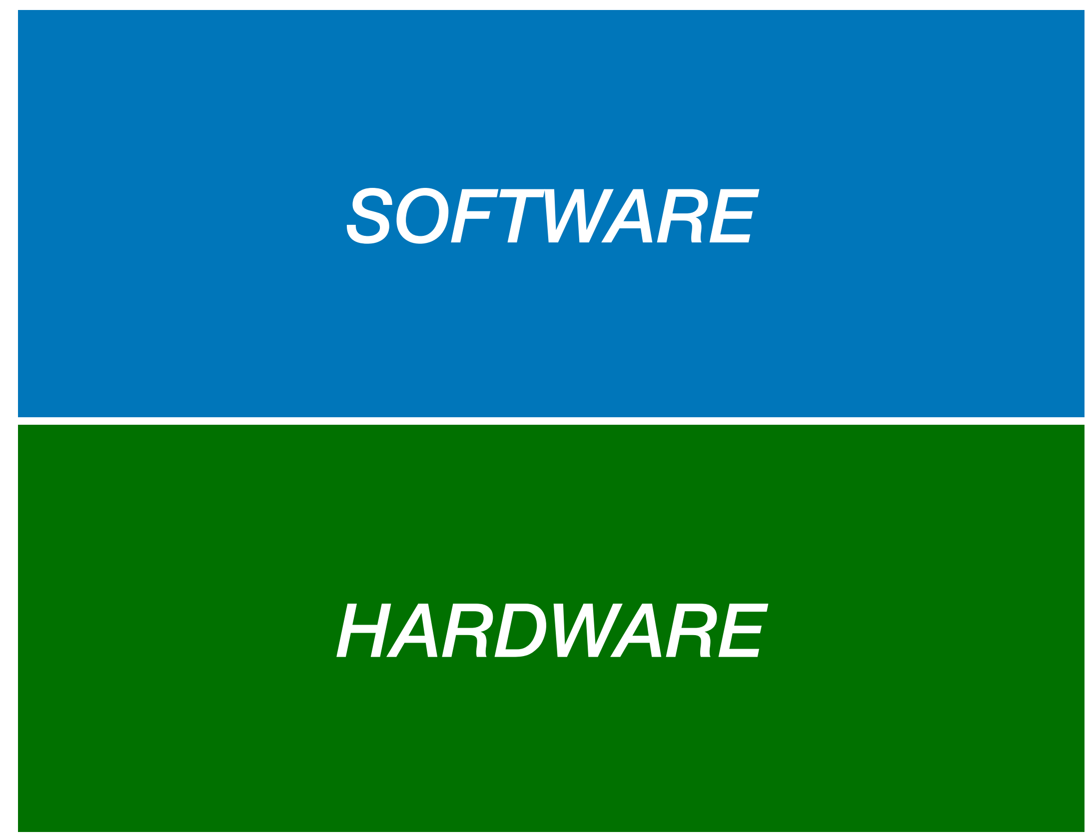
Princípios da Computação

Software

Instituto Superior de Engenharia do Porto

How do computers solve problems?

- **Hardware** is the physical machine that provides a platform for computation.
- **Software** is the set of instructions that can be stored and executed by the hardware.
  - Hard: difficult to modify.
  - Soft: easy to modify.



How to instruct a processor?

- A processor implements an **instruction set architecture (ISA)**.
- The ISA defines an interface between software and hardware, specifying an abstract model of a computer:
  - the execution model,
  - the supported address and data formats,
  - the processor registers,
  - the **instruction set**, i.e. the set of machine instructions that form the machine language.

Implementation of an ISA

- A processor implements an ISA by combining digital logic that allows an instruction set to be executed.
  - The logical design of all electronic components and data paths present in the microprocessor is called **microarchitecture** or **computer organisation**.
- The same ISA can be implemented in different ways (for performance, price, backward compatibility, etc.).
  - The same machine code can be executed by different processors that implement the same ISA (e.g. x86-64).

Machine language instructions

- Machine instructions are encoded into a group of bytes:
  - the operation code — **opcode** — that indicates the operation to be executed,
  - **operands**, indicating which registers/addresses or literal data to be operated.

opcode	**1st operand**	**2nd operand**
01	00A4	0E5F

- Opcodes for a given instruction set are described by an opcode table detailing all possible opcode bytes.

Low vs. high level programming languages

- Writing a program in machine code is not impossible… but very hard and error prone!
- Higher level programming languages provide relevant advantages:
  - Easier for humans to express and understand the program logic.
  - Provide a machine-independent level of abstraction.
- However, it becomes necessary to translate **source code** to **machine code**!

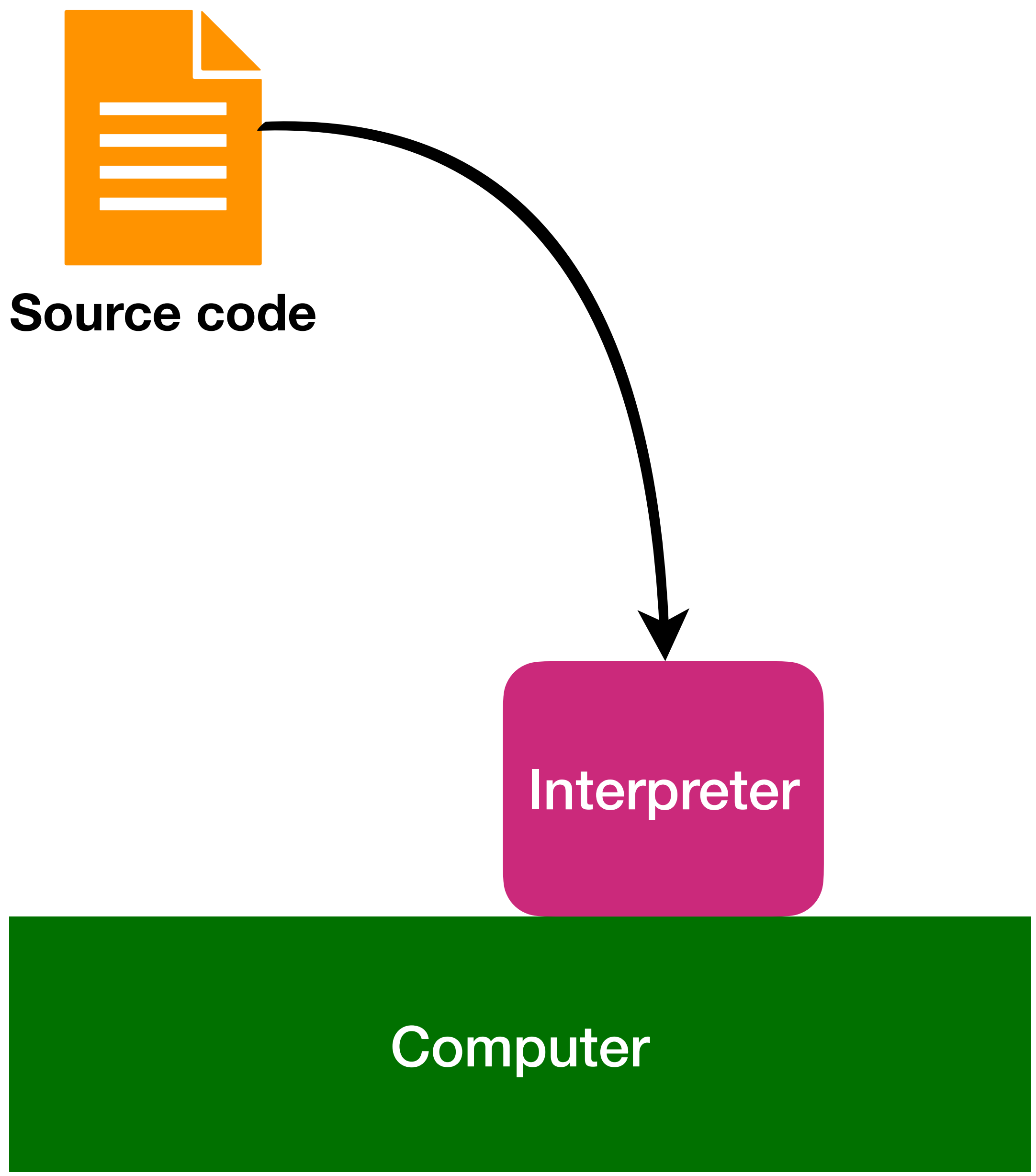
```
#!/usr/bin/env python3

string1 = "PRCMP - "
string2 = "ISEP"
joined_string = string1 + string2
print(joined_string)
```

Interpreted vs. Compiled programs

- Once we have a program written in a high level language, how do we execute it?
- There are two possible approaches:
  - To interprete the program
  - To compile the program

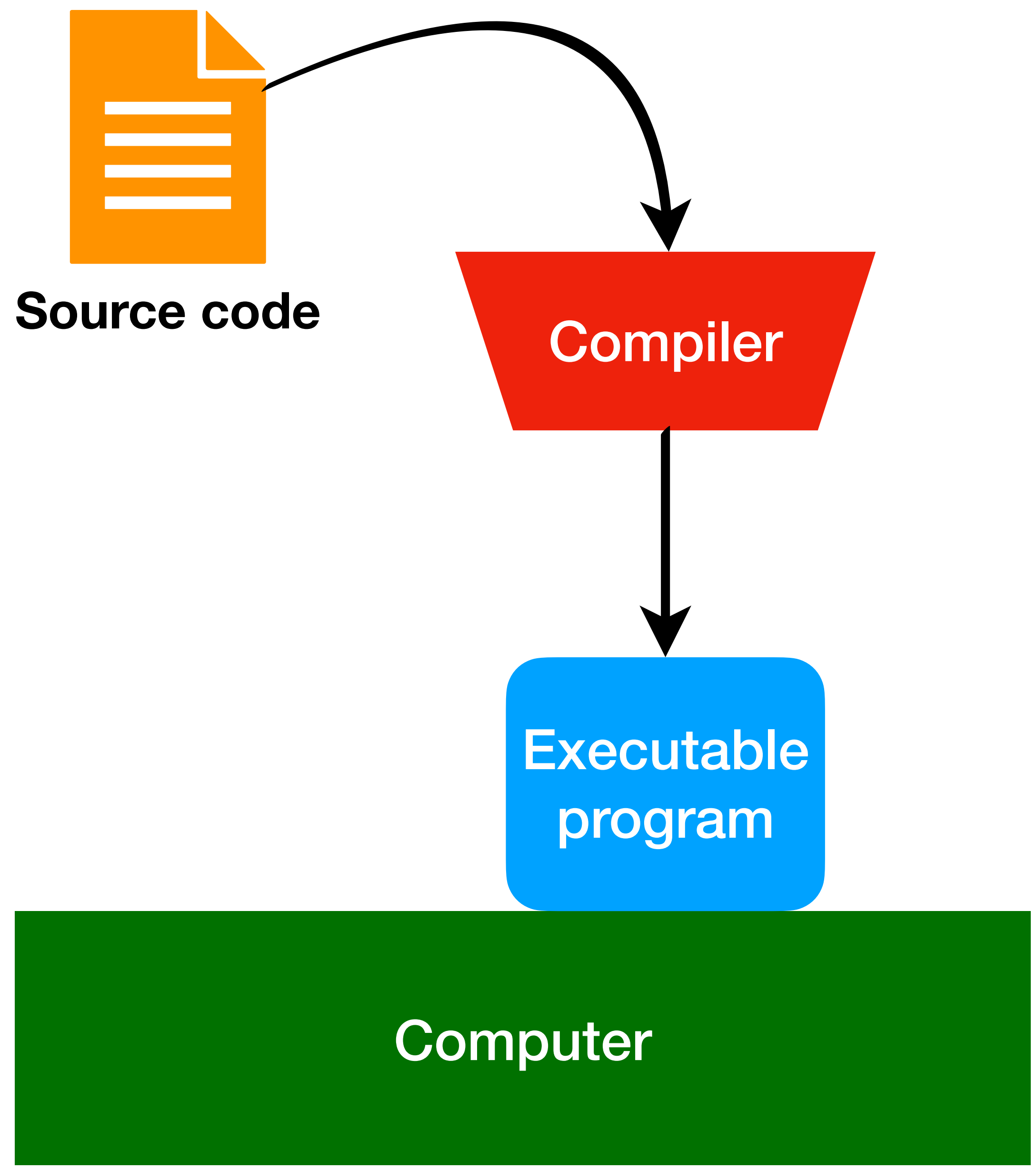
Interpreted programs

- An **interpreter** is a program that executes the instructions in a **script** (source code) file, without producing an executable file.
  - The interpreter creates an environment for the program to execute.
  - The interpreter reads the source file and searches for **keywords**.
  - Iteratively, for each instruction in the source code, the interpreter immediately executes equivalent actions.
- **Translation and execution are interlaced processes!**



Compiled programs

- A **compiler** is a program that translates the source code of an entire program, generating a permanent machine code file.
- This machine code file can be loaded and executed directly by the processor.
- **Translation and execution are separate processes!**



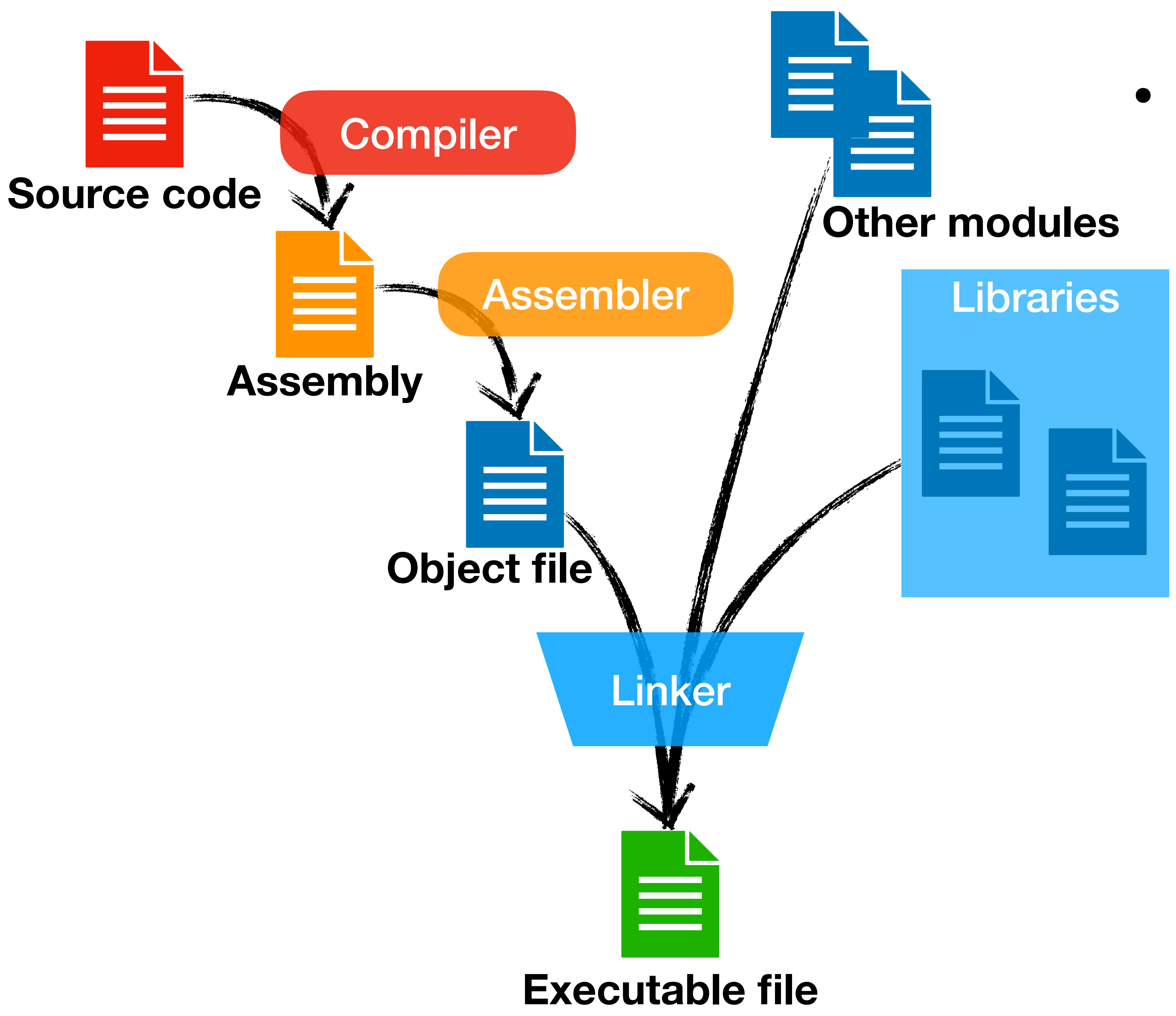
Interpreted vs. Compiled programs

- Both approaches are commonly used!
  - Interpreted: shell scripts, python, javascript, Matlab…
  - Compiled: C, C++, Fortran, java, Swift…
- Both approaches have in common that translate high level instructions down to machine code that is, eventually, executed by the microarchitecture.

Typical steps of a compilation process

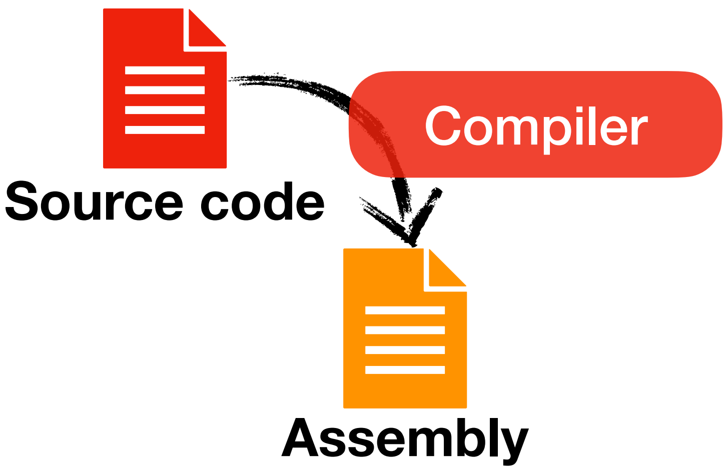
- Building an executable requires to translate high level instructions into machine code instructions.
- But most of the times, it is also necessary to merge code from different modules and libraries to obtain a self-contained executable file.

Typical steps of a compilation process



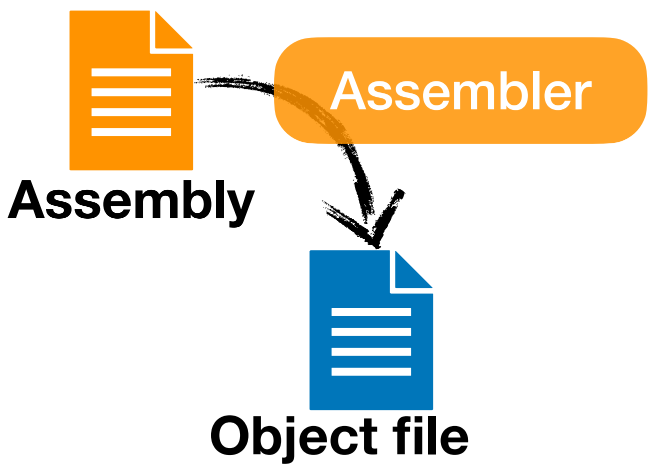
- The typical steps are (in order):
    1. Compilation
    2. Assembly
    3. Linkage

Compilation



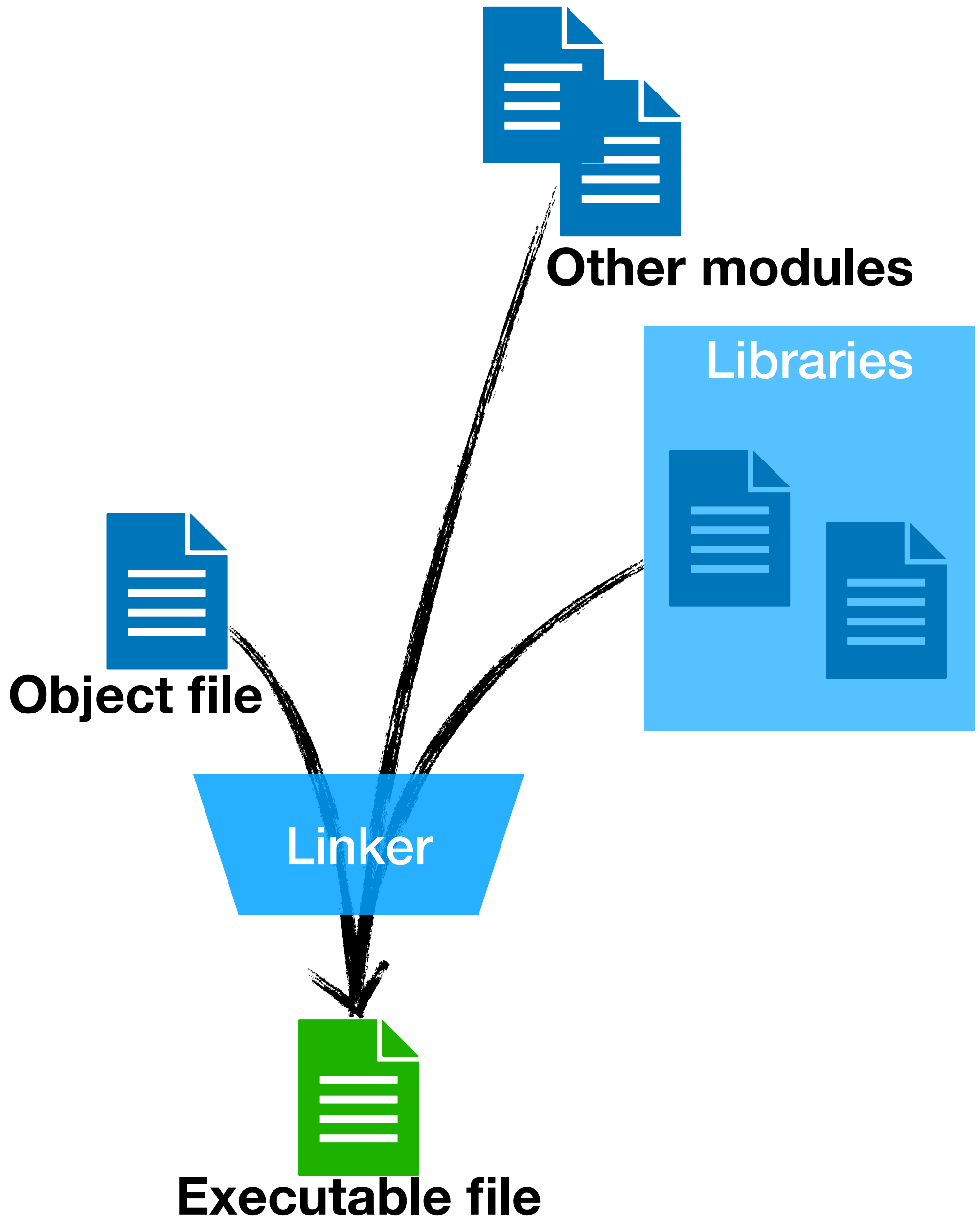
- The compiler translates source code into **assembly language**.
- Assembly language is a human-readable language, that is specific of the target ISA.
  - It can be directly translated to machine code instructions.
- After compilation, the generated assembly keeps symbolic information (names of variables, functions, etc.).

Assembly



- The assembler directly converts the assembly code into (binary) machine code, generating the **object file.**
    - All symbols that can be resolved are substituted by the actual addresses.
    - Symbols that cannot be resolved (e.g. external variable and external function addresses) are kept in the table of symbols.
- As such, the object file is not yet executable!

Linkage



- The linker merges all the object code from multiple modules into a single one.
- If the program is using a function from libraries, linker will link the code with that library function code.
- The linker resolves all remaining symbols replacing by their actual addresses.
- The final file is **executable**!

Cross compilation

- Compile a program for a different architecture.
- May be an architecture with restrictions on resources:
  - Low performance platform, minimalistic architecture, etc.
- Compiling toolchain includes compiler and library for the destination architecture.
  - Program is built on a more powerful / resourceful platform and deployed on the destination architecture.

Memory layout

Memory Layout of an Executing Program

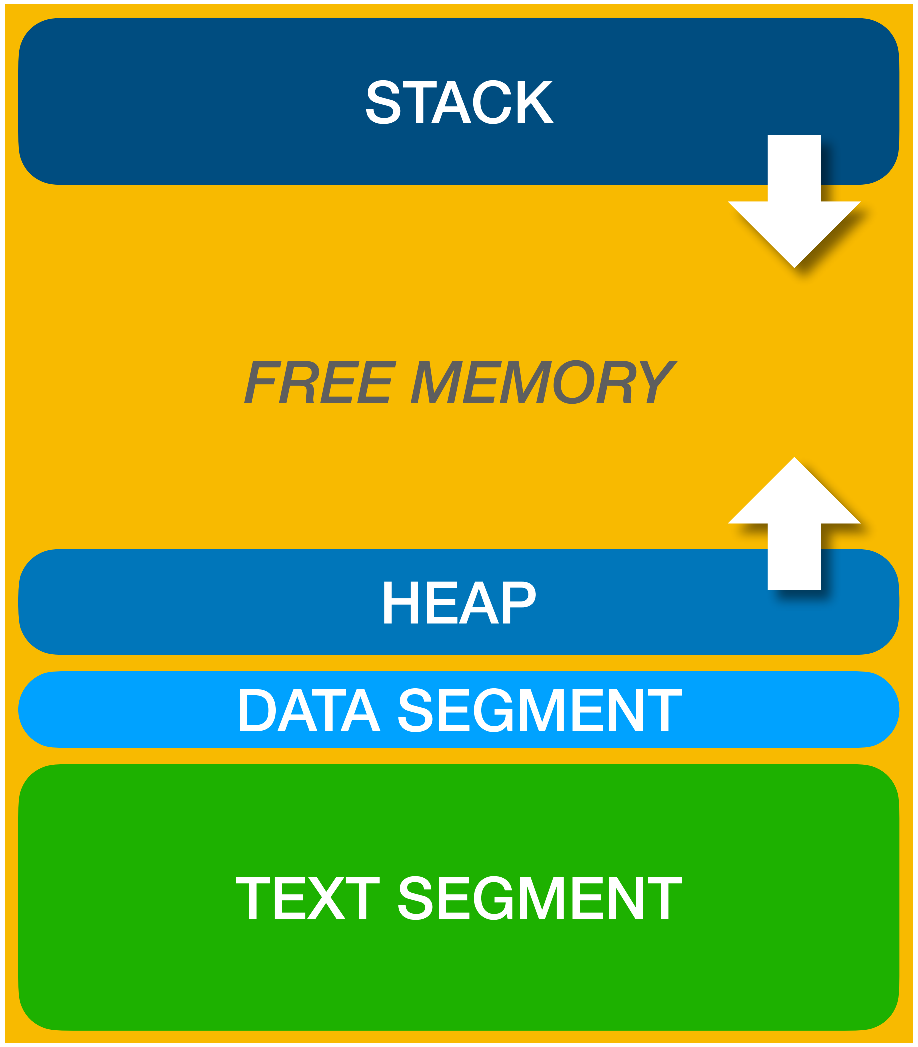
- A running program requires memory for:
  - **instructions:** the program code
  - **data:** global, local and dynamic variables.
- Instructions and data are stored in separate memory regions: the text and data sections.

Text section

- The **Text Segment** contains the executable instructions of the program.

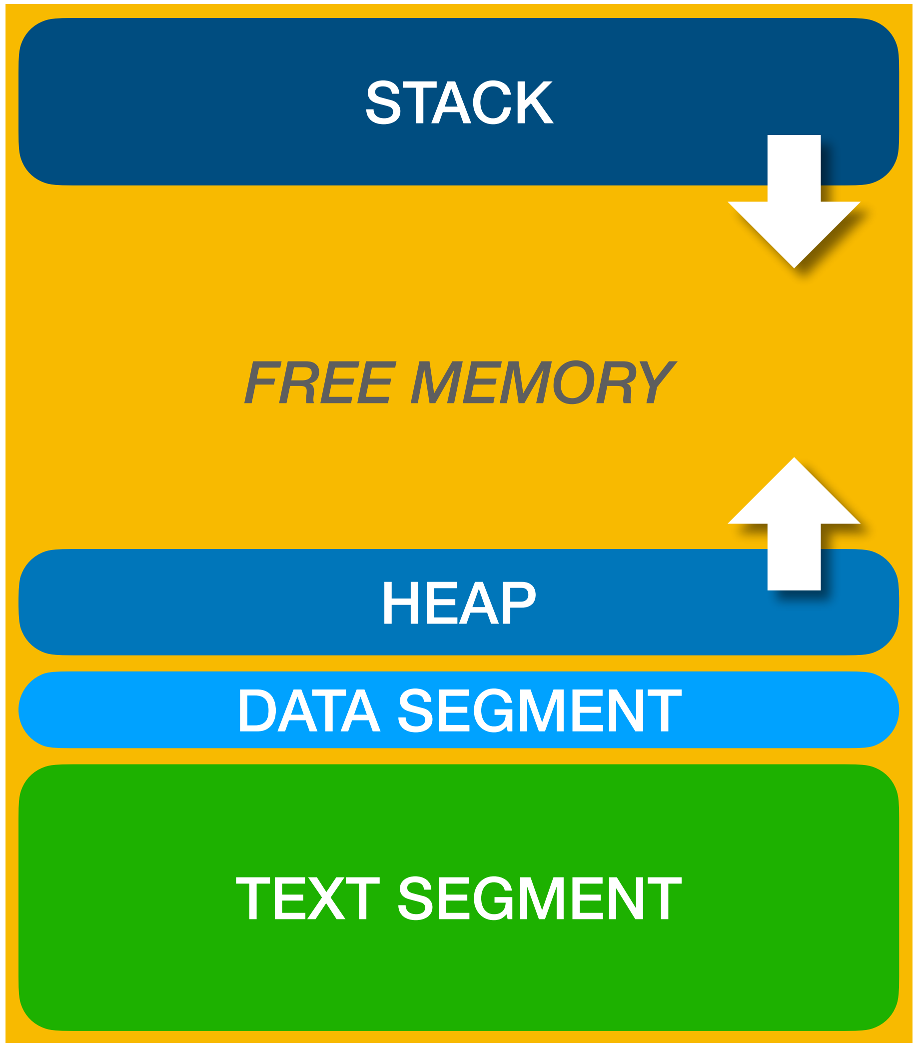
- This segment has fixed size, and is read-only to prevent accidental or malicious modifications.

- Located in **ROM** (firmware) or **RAM** (when the program is loaded from a file).



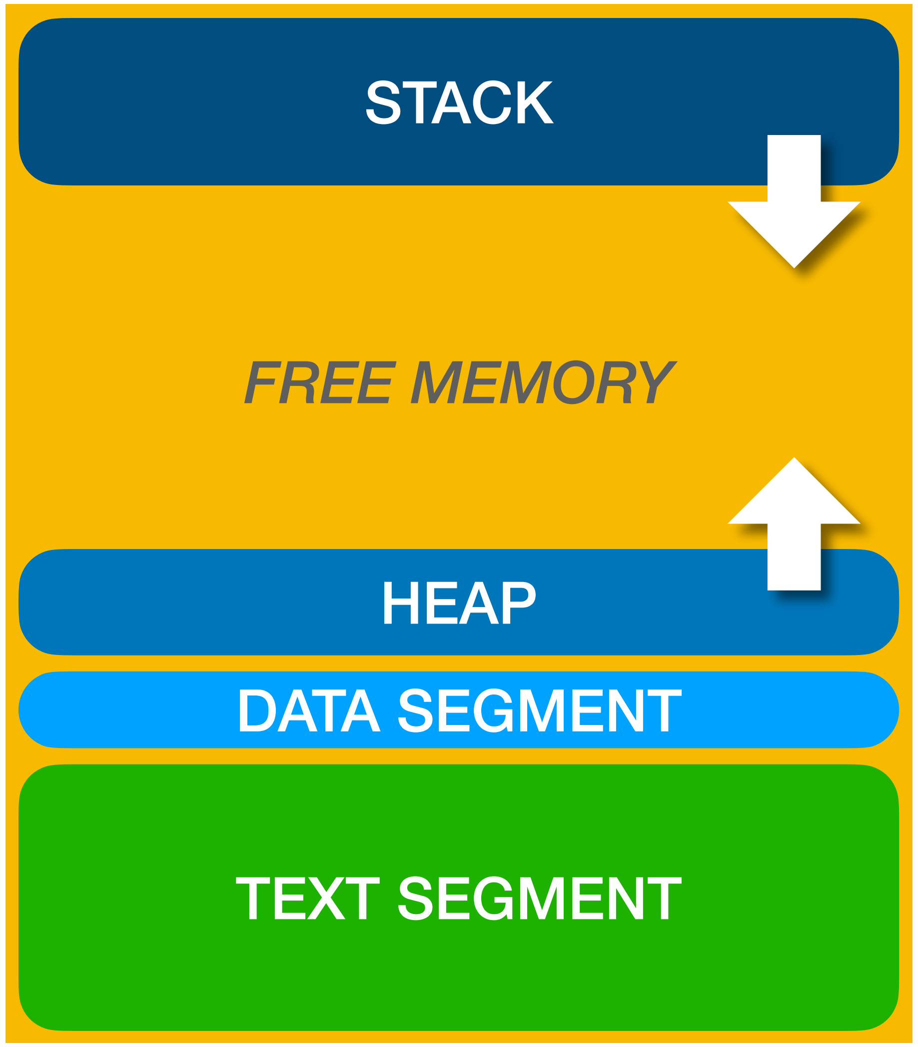
Data section

- The data section (located in **RAM**) is divided into three segments:
- **Data Segment:** holds global and static variables
- **Stack:** stores function calls and local variables
- **Heap:** used for dynamically allocated memory



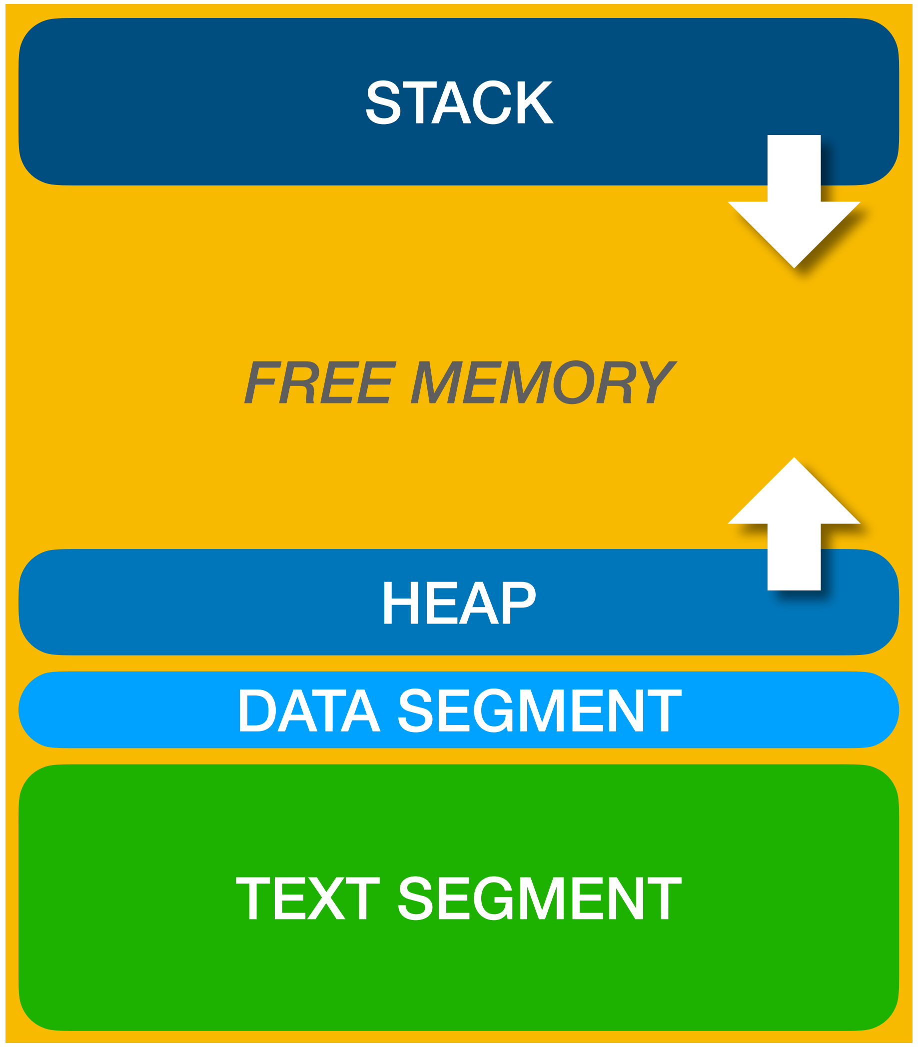
Data Segment

- Global and static variables are determined at compile time and persist throughout the program's execution.
- As a result, the Data Segment has a fixed size for the duration of the program's runtime.



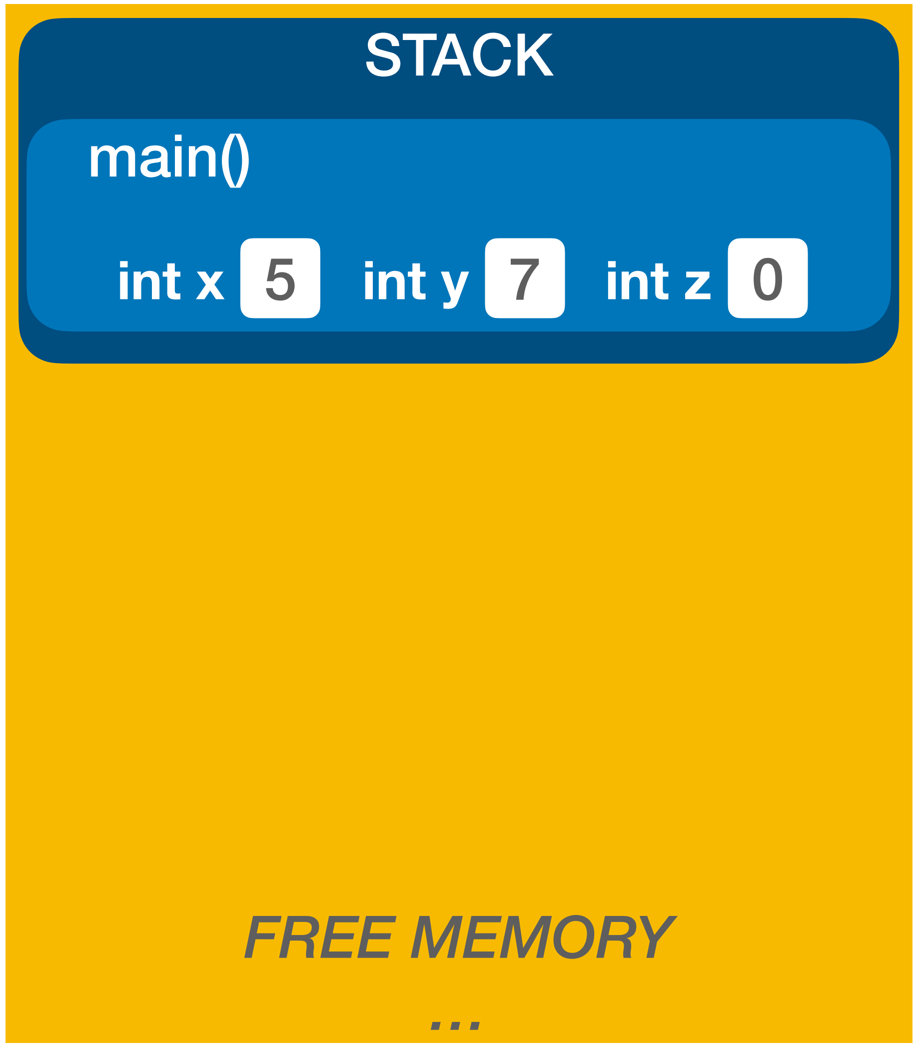
Stack

- The Stack Segment stores function parameters, local variables, and return addresses.
- The Stack shrinks and expands with function calls.



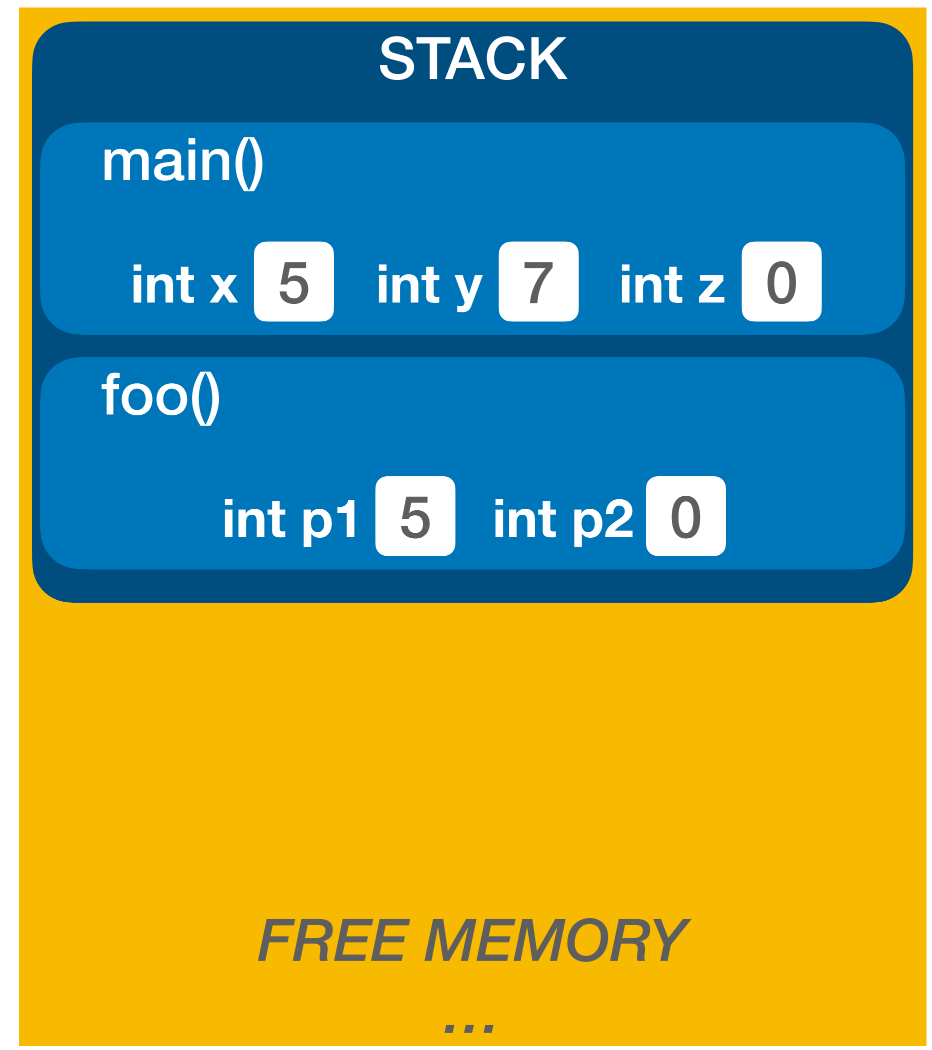
Stack before calling the `foo` function

```
int foo(int p1, int p2)
{ ...
  return;
}

int main()
{
  int x, y, z;
  x = 5;
  y = 7;
  z = 0;
  foo(x, z);
  ...
}
```



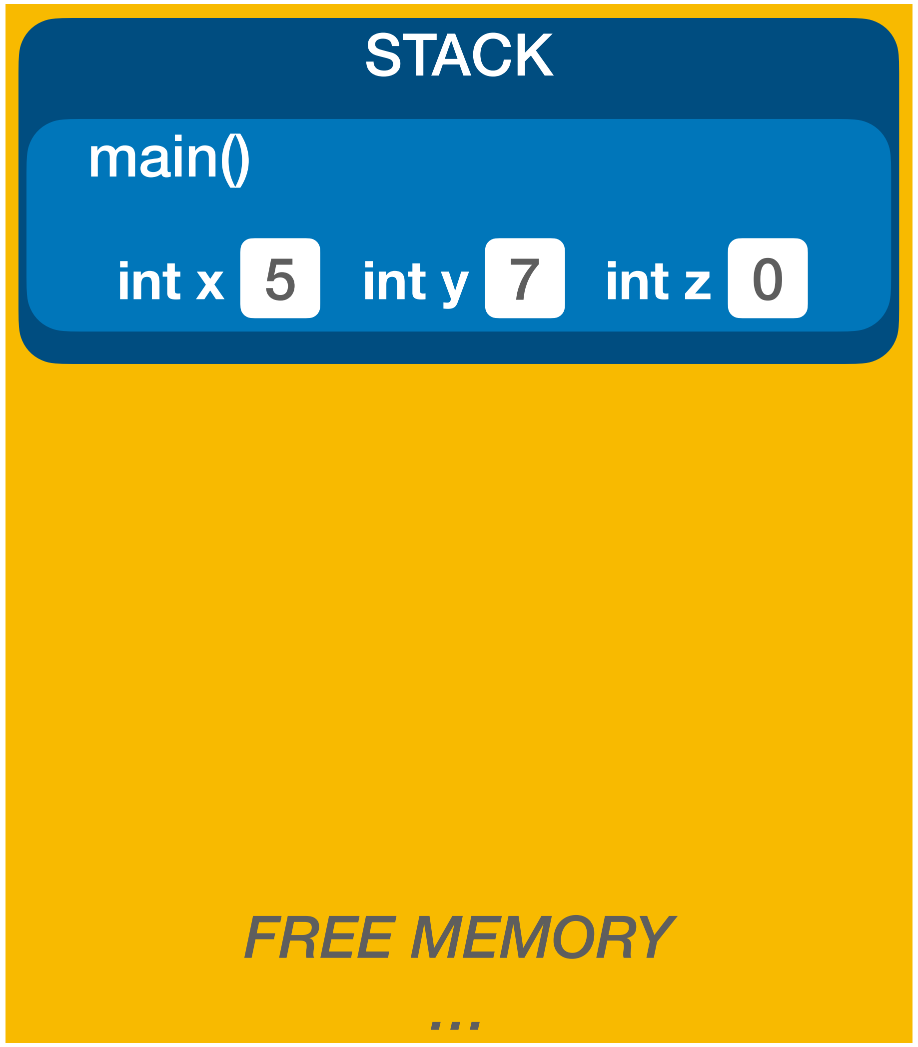
Stack while in the `foo` function

```
int foo(int p1, int p2)
{ ...
  return;
}

int main()
{
  int x, y, z;
  x = 5;
  y = 7;
  z = 0;
  foo(x, z);
  ...
}
```



Stack after returning from the `foo` function

```
int foo(int p1, int p2)
{ ...
  return;
}

int main()
{
  int x, y, z;
  x = 5;
  y = 7;
  z = 0;
  foo(x, z);
  ...
}
```



Heap

- The **Heap Segment** allows for dynamic memory allocation at runtime, which is particularly useful for:
    - **Data Structures:** Ideal for structures whose size or type is not known until runtime.
    - **Arrays:** Supports arrays of unknown size determined at runtime.
    - **Object Instantiation:** Facilitates the creation of objects dynamically during program execution.
- This flexibility enables efficient memory usage and supports complex data management.