Systems Architecture

4. Modular programming in C

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2025/2026

uc3m | Universidad Carlos III de Madrid

Table of contents

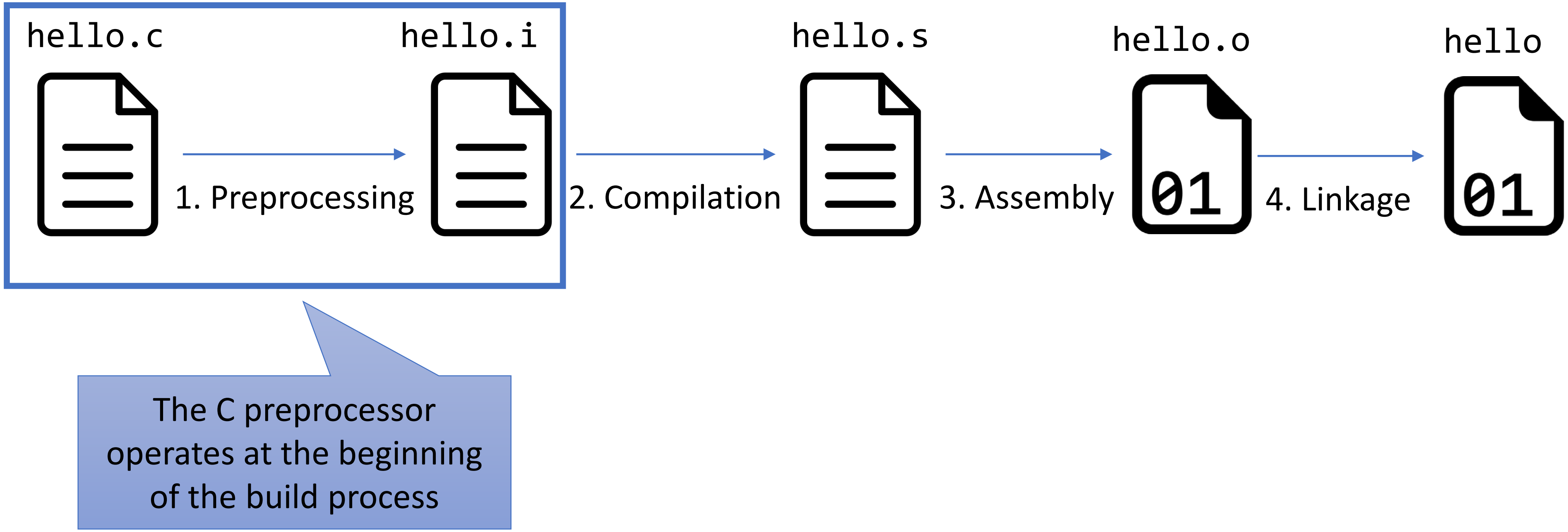
1. Introduction

- So far, we have done C programs with all the logic inside the same source file (e.g., `my-program.c`)
- As C programs grow larger and larger, monolithic programs become difficult to maintain, test, and debug
- For this reason, it is often desirable to split the source code into different files (called **modules**)
- Modularity is important in C programming because it promotes code readability, reusability, maintainability, and flexibility

Table of contents

2. The preprocessor

- The **C preprocessor** is a tool used automatically by the C compiler to transform the program before actual compilation



2. The preprocessor

- **Preprocessor directives** are lines included in the code of programs preceded by a hash sign (**#**)
- The preprocessor examines the code and resolves all these directives before actual compilation
- So far, we have seen a couple of preprocessor directives

```
#include <standard_c_lib.h>
```

To use some C standard library, such as `stdio.h`, `stdlib.h`, etc.

```
#define MACRO value
```

To declare a constant value

2. The preprocessor

- The C preprocessor also allows **conditional compilation** through the following directives:

```
#ifdef MACRO
    /* Code block 1 */
#else
    /* Code block 2 */
#endif
```

If **MACRO** is defined, the first code block is included for compilation. Otherwise, the second block is included

- There is a second directive for conditional compilation called **`#ifndef`**, which is used typically for modular programming

Fork me on GitHub

2. The preprocessor

- Let’s consider the following example:

```
#include <stdio.h>

int main() {
    printf("Hello world\n");

#ifdef DEBUG
    fprintf(stderr, "This is a debug message\n");
#endif

    return 0;
}
```

By default, this message will not be displayed, since **DEBUG** is not defined in this program

```
$ gcc debug_1.c && ./a.out
Hello world
```

2. The preprocessor

- GCC allows defining macros in the command line using the option –D

```
$ gcc -Dname [options] [source files] [-o output file]
$ gcc -Dname=definition [options] [source files] [-o output file]
```

- This way, the previous example displays the debug message if we define the macro DEBUG in the compilation command:

```c
#include <stdio.h>

int main() {
    printf("Hello world\n");

#ifdef DEBUG
    fprintf(stderr, "This is a debug message\n");
#endif

    return 0;
}
```

```
$ gcc debug_1.c && ./a.out
Hello world
```

```
$ gcc debug_1.c -DDEBUG && ./a.out
Hello world
This is a debug message
```

Fork me on GitHub

2. The preprocessor

- In addition to constants, the directive **#define** also allows to create macros with arguments

```
#define MACRO(arguments) expression
```

- These macros work like regular functions in C. For instance:

```c
#include <stdio.h>

#ifdef DEBUG
#define debug(msg) fprintf(stderr, msg)
#else
#define debug(msg)
#endif

int main() {
    printf("Hello world\n");
    debug("This is a debug message\n");
    return 0;
}
```

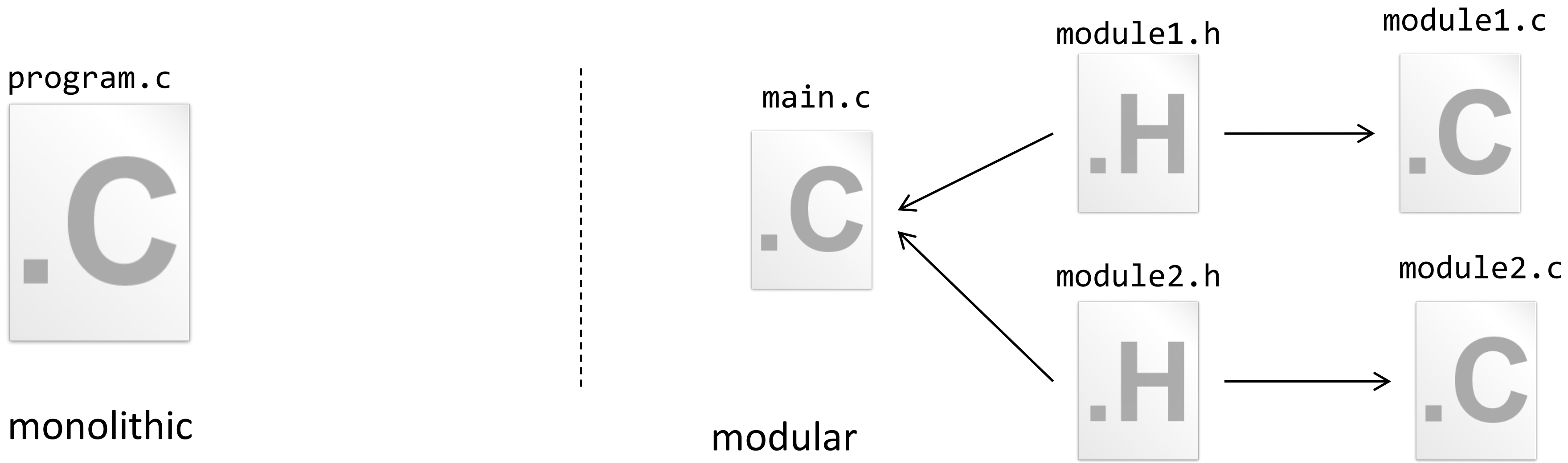
```
$ gcc debug_2.c && ./a.out
Hello world
```

```
$ gcc debug_2.c -DDEBUG && ./a.out
Hello world
This is a debug message
```

Table of contents

3. Modularity

- For implementing modules in C, we need to separate the logic in two different files:
  - **Header** files (.h), which contains functions declarations, global structures, and macro definitions to be shared between several source files (.c)
  - **Source** files (.c) which contains the function definitions



Fork me on GitHub

3. Modularity

- We are going to study modularity through several examples. Consider the following monolithic program that we want to convert in modular

program.c

```
#include <stdio.h>

#define MAX_STR 80

typedef struct Person {
    char name[MAX_STR];
    int age;
} Person;

int sum_ages(Person a, Person b);

int main() {
    Person alice = { "Alice", 25 };
    Person bob = { "Bob", 32 };

    printf("Alice and Bob has %d years together\n", sum_ages(alice, bob));
    return 0;
}

int sum_ages(Person a, Person b) {
    return a.age + b.age;
}
```

3. Modularity

- We want to separate the declarations and macro definitions to a header file (.h), and the functions definitions to a source file (.c)

```
#include <stdio.h>

#define MAX_STR 80

typedef struct Person {
    char name[MAX_STR];
    int age;
} Person;

int sum_ages(Person a, Person b);

int main() {
    Person alice = { "Alice", 25 };
    Person bob = { "Bob", 32 };

    printf("Alice and Bob has %d years together\n", sum_ages(alice, bob));
    return 0;
}

int sum_ages(Person a, Person b) {
    return a.age + b.age;
}
```

Macro definition

Structure declaration

Function declaration

.H

Function definition

.C

3. Modularity

Fork me on GitHub

main.c

```c
#include <stdio.h>
#include "person.h"

int main() {
    Person alice = { "Alice", 25 };
    Person bob = { "Bob", 32 };

    printf("Alice and Bob has %d years together\n",
            sum_ages(alice, bob));
    return 0;
}
```

Notice that the directive **#include** also allows to include custom header files (when using " ")

#ifndef and #define are known as ***header guards***. Their primary purpose is to prevent header files from being included multiple times

person.h

```c
#ifndef PERSON_H
#define PERSON_H

#define MAX_STR 80

typedef struct Person {
    char name[MAX_STR];
    int age;
} Person;

int sum_ages(Person a, Person b);

#endif
```

person.c

```c
#include "person.h"

int sum_ages(Person a, Person b) {
    return a.age + b.age;
}
```

3. Modularity

- GCC allows compilating separately the modules, and then a linkage the resulting object files into a single binary file
  - For instance, in the example before:

```
gcc main.c -c
```

The flag `-c` compiles and assemble, but do not link. In this example, it only produces `main.o`

```
gcc person.c -c
```

It only produces `person.o`

```
gcc main.o person.o -o main
```

It links `main.o` and `person.o`, producing the executable program (called `main` in this example

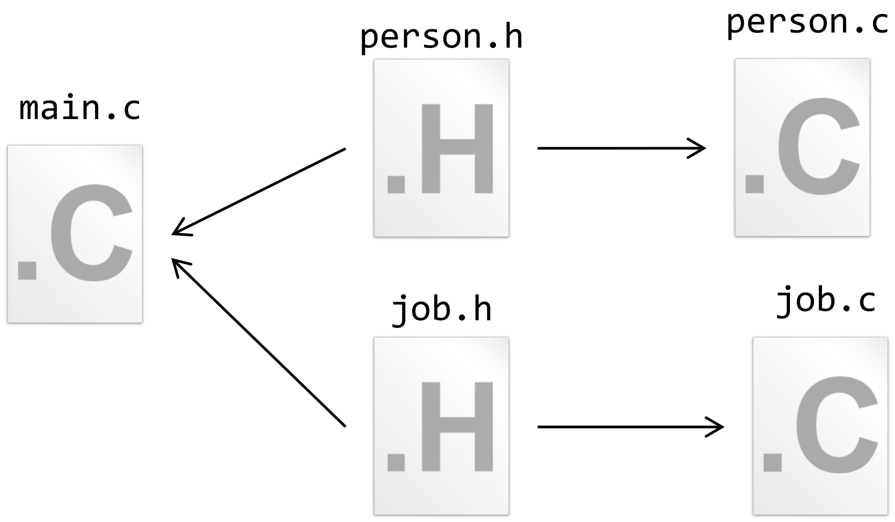
- To simplify, and supposing that all modules of our program belong to the same folder, we can compile and linkage all modules using a single command

```
gcc *.c -o main
```

It produces the executable program with a single command. This command assumes all source files (`*.c`) are in the same folder

3. Modularity

- To see the importance of header guards, let's consider now another example of a program composed of two modules:



Fork me on GitHub

3. Modularity

main.c

```c
#include <stdio.h>
#include "person.h"
#include "job.h"

int main() {
    Person alice = { "Alice", 25 };
    Person bob = { "Bob", 32 };

    Job developer = { alice, "developer" };
    Job tester = { bob, "tester" };

    display_job(developer);
    display_job(tester);

    return 0;
}
```

person.h

```c
#ifndef PERSON_H
#define PERSON_H

#define MAX_STR 80

typedef struct Person {
    char name[MAX_STR];
    int age;
} Person;

int sum_ages(Person a, Person b);

#endif
```

job.h

```c
#ifndef JOB_H
#define JOB_H

#include "person.h"

typedef struct Job {
    Person person;
    char role[MAX_STR];
} Job;

void display_job(Job job);

#endif
```

```
In file included from job.h:4,
                 from main.c:3:
person.h:4:16: error: redefinition of ‘struct Person’
    4 | typedef struct Person {
      |                ^~~~~~
In file included from main.c:2:
person.h:4:16: note: originally defined here
    4 | typedef struct Person {
      |                ^~~~~~
```

Without *header guards*, we will get compilation errors like this:

Fork me on GitHub

3. Modularity

- When using global variables, we need to use the keyword **extern** in the variables defined in other module:

main.c

```c
#include <stdio.h>
#include "person.h"
#include "job.h"

Job company[MAX_JOBS];

int main() {
    Person alice = { "Alice", 25 };
    Person bob = { "Bob", 32 };

    Job developer = { alice, "developer" };
    Job tester = { bob, "tester" };

    company[0] = developer;
    company[1] = tester;

    display_job_by_index(0);
    display_job_by_index(1);

    return 0;
}
```

job.c

```c
#include <stdio.h>
#include "job.h"

extern Job company[];

void display_job(Job job) {
    printf("%s is a %s\n", job.person.name, job.role);
}

void display_job_by_index(int i) {
    display_job(company[i]);
}
```

3. Modularity

- Common **bad practices** in modular programming in C are:
  - Include global variables in headers files

person.h

```c
#ifndef JOB_H
#define JOB_H

typedef struct Job {
    Person person;
    char role[MAX_STR];
} Job;

Job company[MAX_JOBS];

#endif
```

This might lead to *multiple definition* errors

  - Include functions definitions in headers file:

job.h

```c
void display_job(Job job) {
    printf("%s is a %s\n", job.person.name, job.role);
}

void display_job_by_index(int i) {
    display_job(company[i]);
}
```

Table of contents

4. Makefile

- The **`make`** tool allows managing and maintaining computer programs consisting in several component files
- The `make` tool reads the instruction defined in a file called **`Makefile`** (also known as descriptor file)
- The Makefile file is composed by a sets a set of rules to determine which parts of a program need to be compiled, how it is executed, or how to clean the intermediate file (e.g. object files)

GNU Make

https://www.gnu.org/software/make/

4. Makefile

- A `Makefile` is made up of different sections, each one containing:
  - Target: Normally, an executable or object file
  - Dependencies: Source code or other targets
  - Rules: Set of commands needed to make the target

Important: every rule line begins with a **tab**, not spaces

```
# Comment
target: dependency
        command_1
        command_2
        ...
        command_N
```

- Also, it is possible to define variables in a `Makefile`:

```
VAR_NAME=value
```

Fork me on GitHub

4. Makefile

- For example (module 1):

```
CFLAGS=-Wall

compile:
	gcc $(CFLAGS) main.c -c
	gcc $(CFLAGS) person.c -c
	gcc $(CFLAGS) main.o person.o -o main

clean:
	rm -f *.o
	rm -f main

run: compile
	./main
```

```
$ make
gcc -Wall main.c -c
gcc -Wall person.c -c
gcc -Wall main.o person.o -o main
```

```
$ make run
gcc -Wall main.c -c
gcc -Wall person.c -c
gcc -Wall main.o person.o -o main
./main
Alice and Bob has 57 years together
```

```
$ make clean
rm -f *.o
rm -f main
```

4. Makefile

- Another example (module 2):

```
CFLAGS=-Wall

compile:
	gcc $(CFLAGS) *.c -o main

clean:
	rm -f main

run: compile
	./main
```

```
$ make
gcc -Wall *.c -o main
```

```
$ make run
gcc -Wall *.c -o main
./main
Alice is a developer
Bob is a tester
```

```
$ make clean
rm -f main
```

Table of contents

Fork me on GitHub

5. Static variables

- Static variables are defined using the keyword **static**
  - These variables are initialized only once
  - Therefore, the compiler persists with the variable till the end of the program

```c
#include <stdio.h>

void my_function() {
    int regular_int = 0;
    static int static_int = 0;

    regular_int++;
    static_int++;

    printf("regular_int = %d, static_int = %d\n", regular_int, static_int);
}

int main() {
    for (int i = 0; i < 10; i++) {
        my_function();
    }
}
```

```
regular_int = 1, static_int = 1
regular_int = 1, static_int = 2
regular_int = 1, static_int = 3
regular_int = 1, static_int = 4
regular_int = 1, static_int = 5
regular_int = 1, static_int = 6
regular_int = 1, static_int = 7
regular_int = 1, static_int = 8
regular_int = 1, static_int = 9
regular_int = 1, static_int = 10
```

5. Static variables

- We can also use the static keyword for implementing **encapsulation** in module (i.e., access restriction):
  - Static global variables are not visible outside of the file they are defined in
  - Static functions are not visible outside of the C file they are defined in

For instance, this variable can only be used in this file (even if other files try to access with **extern**)

```c
#include <stdio.h>
#include "person.h"
#include "job.h"

static Job company[MAX_JOBS];

int main() {
    // ...

    return 0;
}
```

Table of contents

6. Takeaways

- The C preprocessor is a used automatically by the C compiler to expand macros (e.g. `#include`, `#define`) or conditional compiling (e.g. `#ifdef`, `#ifndef`)
- GCC allows defining macros in the command line using the option -D (e.g., for debugging)
- For modular programs in C, we need to separate the logic into headers (.h) and source (.c) files
- Header files (.h) will contain functions declarations, global structures, and macro definitions, while source files (.c) will contain the function definitions
- The `make` tool reads the instructions defined in a file called `Makefile` (also known as descriptor file) to compile, execute or clean C programs
- Static variables (defined with the keyword `static`) in C are initialized only once