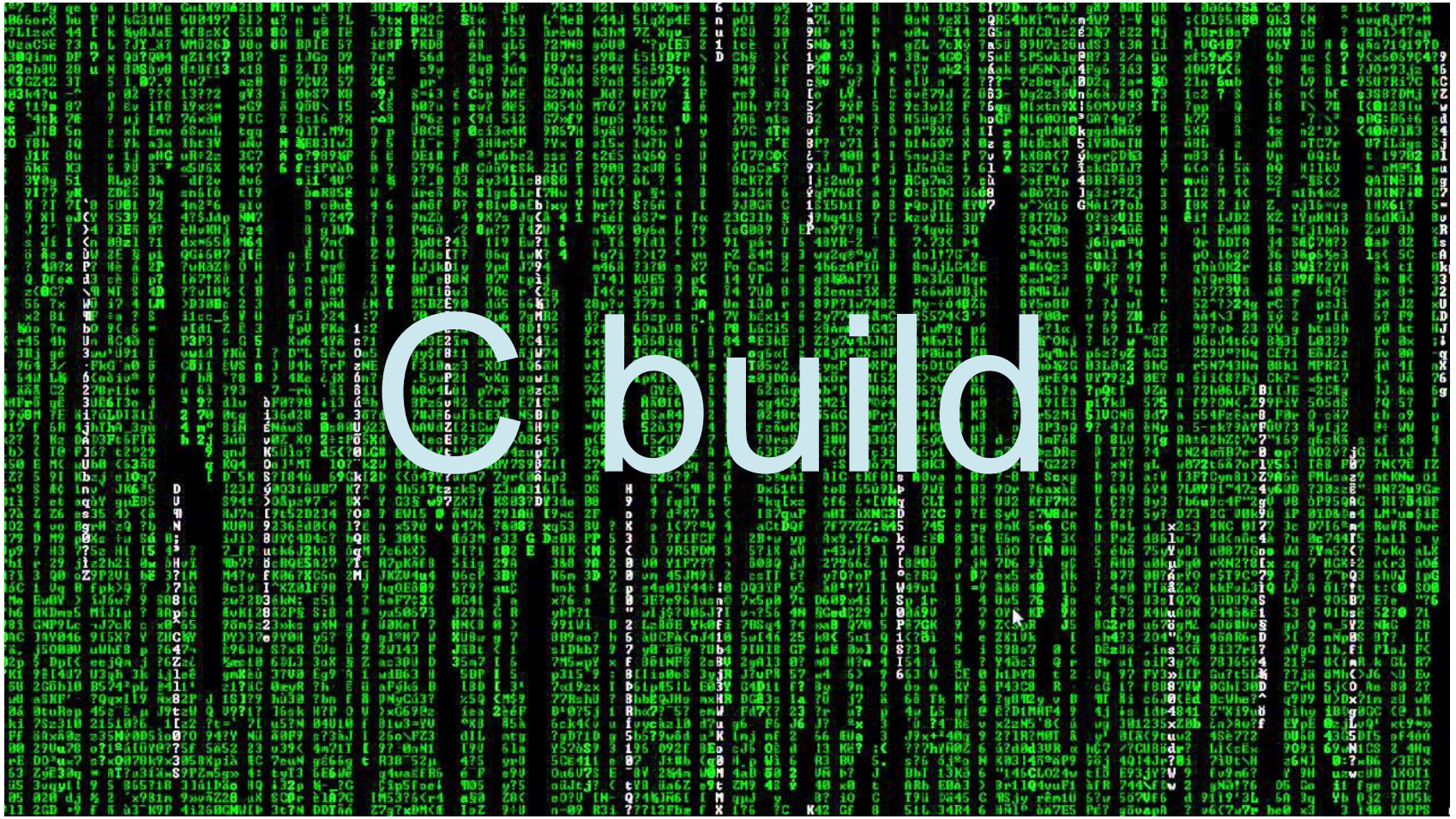
ESIGELEC
ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

# C Programming




Ángel Perles
angel.perles@upv.es
R06

# Contents

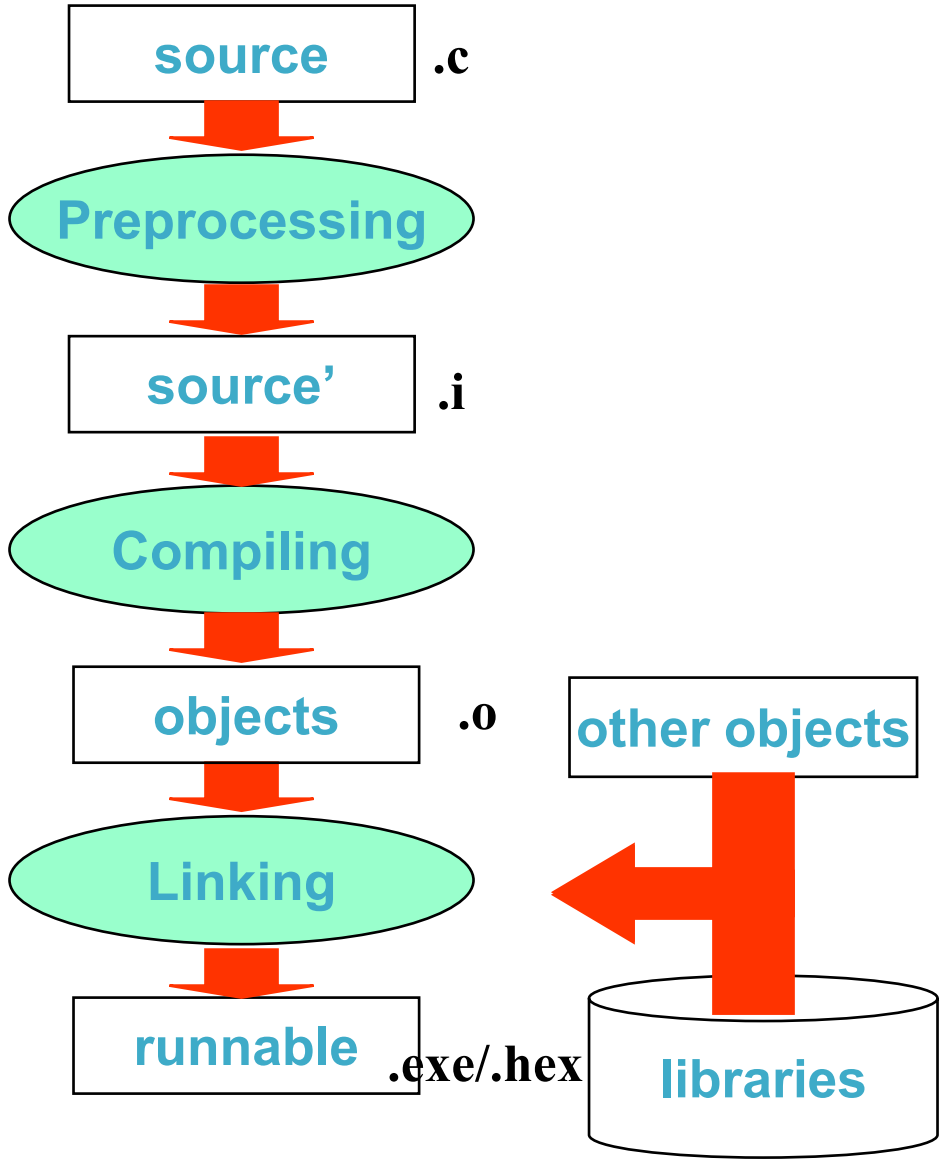
Objectives
C build stages
An example
Building it using GCC
Seeing the stages with GCC
Splitting into modules
A template for C files
Why compile separately?
Which stage failed?
Automating the build: make
Comparing C and Java

# Objectives

- By the end of this session you should be able to:
  - Describe the build stages and what each one produces
  - Build single- and multi-file programs with *gcc* and the course flags
  - Write a header file with include guards and use it correctly
  - Tell whether an error comes from the **compiler** or the **linker**
  - Explain how *make* decides what to rebuild

# C build stages

- "Building" a C project steps
  - "Preprocessing" C source files
  - "Compiling" preprocessed C files
  - "Linking" the compiled files and external libraries
- **Note:** "compiling" is only one stage; "building" is the whole chain



# C build stages

- "Building" a C project steps
  - The **preprocessor** replaces text driven by *#keywords*

```
/* main.c */
#include <stdio.h>
#define PI 3.14159265358979323846

double area_circle(double radius);

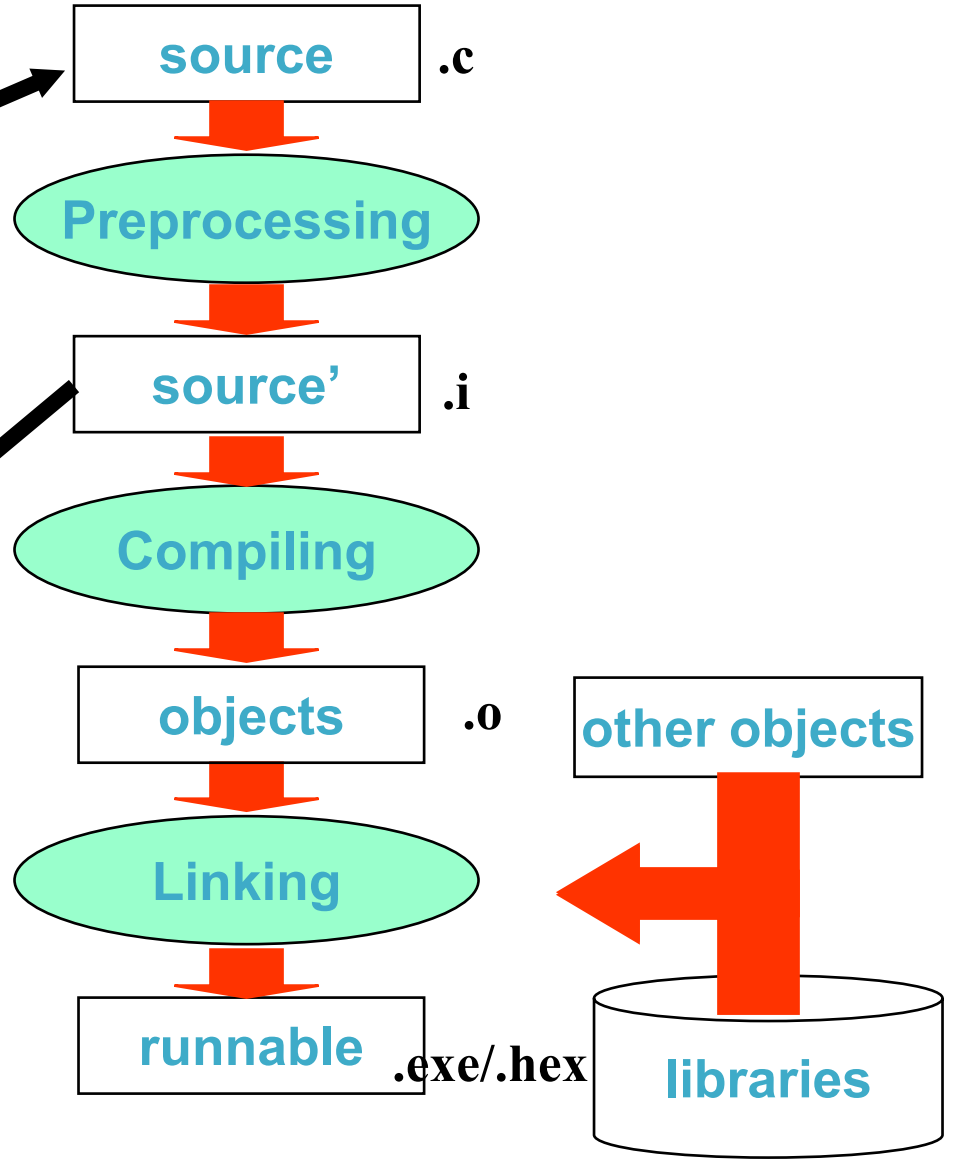
int main(void) { ... }

double area_circle(double radius)
{
    return PI * radius * radius;
}
```

```
/* main.i (preprocessed) */
/* ...~800 lines from stdio.h */
int printf(const char *format, ...);
int scanf(const char *format, ...);
double area_circle(double radius);

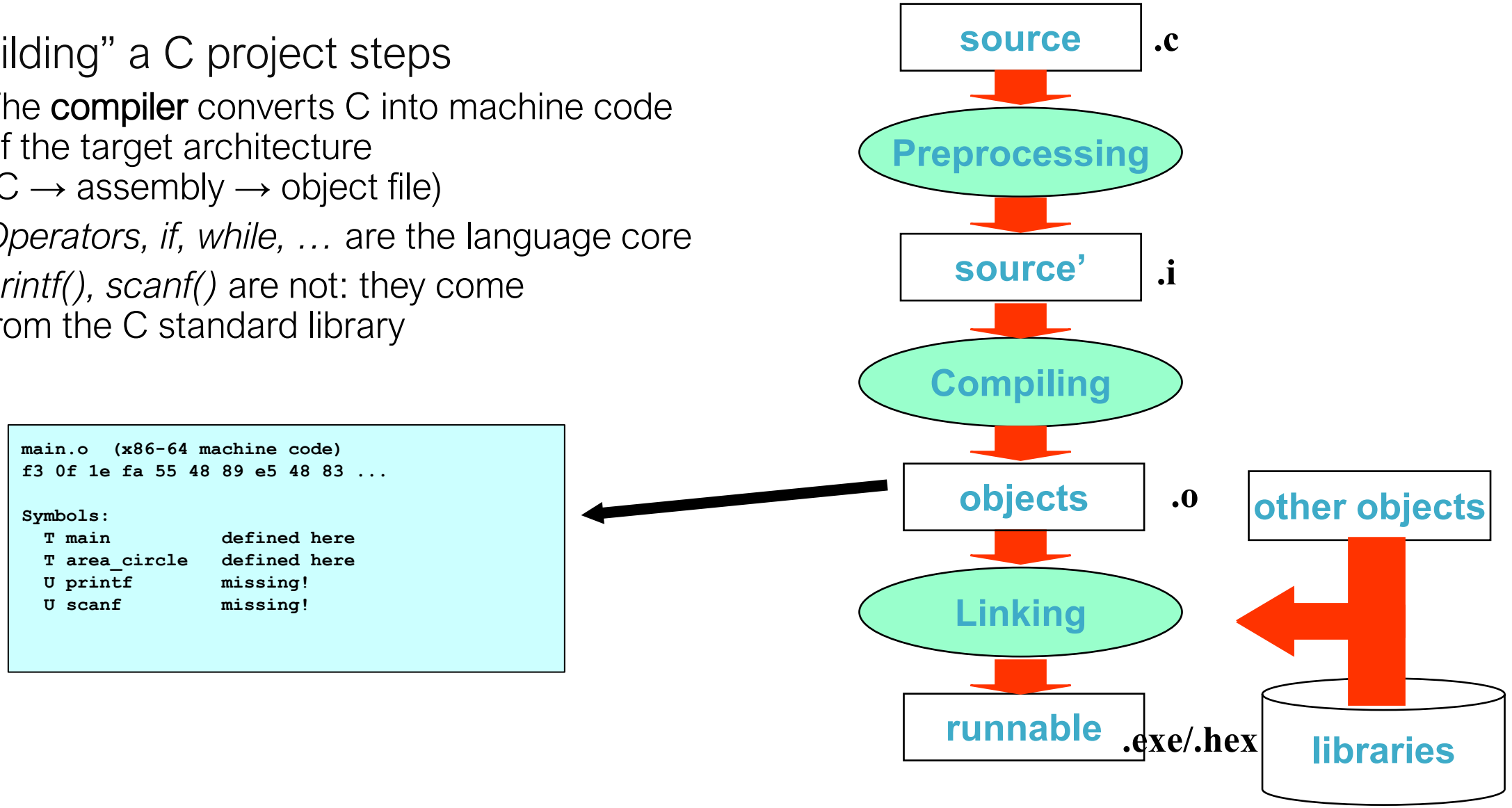
int main(void) { ... }

double area_circle(double radius)
{
  return 3.14159265358979323846
         * radius * radius;
}
```



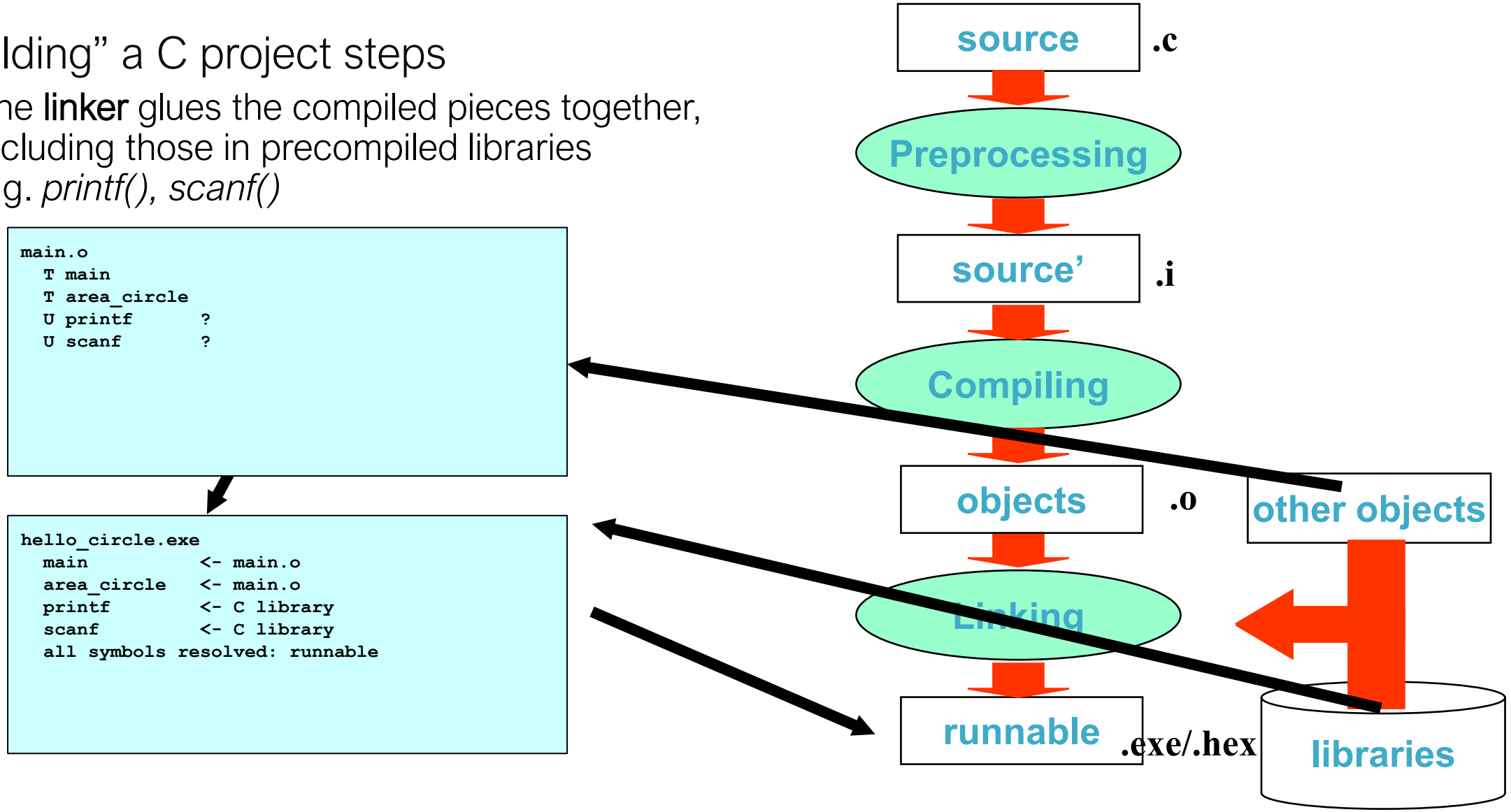
# C build stages

- "Building" a C project steps
  - The **compiler** converts C into machine code of the target architecture (C → assembly → object file)
  - *Operators, if, while, …* are the language core
  - *printf(), scanf()* are not: they come from the C standard library



# C build stages

- "Building" a C project steps
  - The **linker** glues the compiled pieces together, including those in precompiled libraries
    e.g. *printf(), scanf()*



# An example

- Minimal example to calculate the area of a circle
  - Preprocessor directives
  - Compilation flow



```c
#include <stdio.h>

#define PI 3.14159265358979323846

double area_circle(double radius);

int main(void)
{
    double radius;

    printf("Enter the circle radius: ");
    scanf("%lf", &radius);
    printf("The area of the circle is %.2f\n", area_circle(radius));

    return 0;
}

double area_circle(double radius)
{
    return PI * radius * radius;
}
```

# Building it using GCC

- To build a C project we need a “toolchain”
  - Compiler, assembler, preprocessor, linker …
  - And extra tools are very welcome (and common)
  - A very popular toolchain set is the open GNU Compiler Collection (GCC)
    - Managed by the Free Software Foundation
    - UNIX-like
    - Thousands of architectures supported
  - Assuming it is installed, let’s invoke it from a terminal to build our “project”

**C:\...\first_project> gcc main.c -o hello_circle -std=c99 -Wall -Wextra -Wpedantic**



# Building it using GCC

- Assuming it is installed, let's invoke it from a terminal to build our "project"

```
C:\...\first_project> gcc main.c -o hello_circle -std=c99 -Wall -Wextra -Wpedantic
```

- Where
  - *first_project* is a disc folder containing our source code
  - *-std=c99* is to tell the *gcc* compiler to use the C99 standard
  - *-Wall -Wextra –Wpedantic* is to be very obsessed with perfection (helps for quality code)
  - *main.c* is our text file containing the C source code
  - *-o hello_circle* is the desired output name for the executable app



# Building it using GCC

- Running this build process, should result in a file called *hello_circle.exe* (for MWindows) that you should be able to run

```
Command Prompt

H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex01>gcc main.c -o hello_circle -st
d=c99 -Wall -Wextra -Wpedantic

H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex01>dir
 Volume in drive H is VBOX_home_aperles
 Volume Serial Number is 0001-0302

 Directory of H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex01

27/10/2026  11:10    <DIR>          .
27/10/2026  11:10            40,499 hello_circle.exe
26/10/2026  13:14    <DIR>          ..
26/10/2026  13:36               376 main.c
               2 File(s)         49,067 bytes
               2 Dir(s)  726,548,901,888 bytes free

H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex01>hello_circle
Enter the circle radius: 69
The area of the circle is 14957.12

H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex01>
```



# Seeing the stages with GCC

- *gcc* can stop after any stage. Try it with *main.c*:

```
gcc -E main.c -o main.i   # preprocess only          -> main.i
gcc -S main.c             # ... and compile          -> main.s (assembly)
gcc -c main.c             # ... and assemble         -> main.o (object)
nm main.o                 # list the symbols inside main.o
```

*(add the course flags -std=c99 -Wall -Wextra -pedantic to the gcc commands)*

## Questions

Open *main.i*: how many lines does it have? Where did *PI* go?
In the output of *nm*, what does *U printf* mean? Who will fix it?

# Splitting into modules

- A real C project should be divided into modules
  - Like any other programming project in Java, Rust, Node.js, Ada, Assembler …
  - Following typical development patterns such as abstraction, decoupling, etc.
- Let's divide the example into logical modules

```c
/* area.h */
#ifndef AREA_H
#define AREA_H

double area_circle(double radius);

#endif
```

```c
/* main.c */
#include <stdio.h>
#include "area.h"

int main(void)
{
    double radius;

    printf("Enter the circle radius: ");
    scanf("%lf", &radius);
    printf("The area of the circle is %.2f\n", area_circle(radius));

    return 0;
}
```

```c
/* area.c */
#include "area.h"

#define PI 3.14159265358979323846

double area_circle(double radius)
{
    return PI * radius * radius;
}
```

# Splitting into modules

- Let's analyze *main.c*
  - *area.h* is a *header file*: C source code containing function prototypes, data type definitions, etc.

```c
/* main.c */
#include <stdio.h>
#include "area.h"

int main(void)
{
    double radius;

    printf("Enter the circle radius: ");
    scanf("%lf", &radius);
    printf("The area of the circle is %.2f\n", area_circle(radius));

    return 0;
}
```

- **Barr-C NOTE**: Now, I can tell you that I'm following a style guide (Barr-C) for names of things, brackets, order of .h file inclusion, etc.

# Splitting into modules

- Let's analyze *area.h*
  - I insist, it is a ***header file***: C source code containing function prototypes, data type definitions, etc.
  - Barr-C rule: every header **must include *guards*** against "multiple inclusion"
  - A guard is a text label that the preprocessor can check

```
1   /* area.h */
2   #ifndef AREA_H
3   #define AREA_H
4
5   double area_circle(double radius);
6
7   #endif
```

- **Barr-C**: Module names are using the prefix of the functions. The guard is using the name of the header file (more or less) capitalized

# Splitting into modules

- Let's analyze *area.c*
  - The module includes its associated header file to avoid misunderstandings …
  - … that is, to warrant consistency
  - It is easy to lie in C and build crappy code … that's real life

```c
/* area.c */
#include "area.h"

#define PI 3.14159265358979323846

double area_circle(double radius)
{
    return PI * radius * radius;
}
```

# A template for C files: the header (.h)

```c
/**
    @file    template.h
    @brief   This file is a template for C header files
    ...
*/
/* Define to prevent recursive inclusion ---*/
#ifndef TEMPLATE_H
#define TEMPLATE_H
#ifdef __cplusplus
 extern "C" {
#endif
/* Includes ------------------------------*/
#include <stdint.h>
/* Exported types ------------------------*/
/* Exported constants --------------------*/
/* Exported macro ------------------------*/
/* Exported functions --------------------*/
uint8_t template_init(uint16_t peters);
#ifdef __cplusplus
}
#endif
#endif /* TEMPLATE_H */
/*** End of file **************************/
```

### Header comment

Fill in the tags (@file, @brief…): documentation tools such as Doxygen read them

### Include guard

Already in place: rename **`TEMPLATE_H`** after your module, e.g. **`AREA_H`**

### extern "C"

Lets C++ code use this module. Leave it as it is

### Exported sections

Types, constants, macros and prototypes: the public contract of the module

# A template for C files: the source (.c)

```c
/**
    @file    template.c
    @brief   This file is a template for C ...
    ...
*/
/* Includes ------------------------------*/
#include "template.h"
#include <stdint.h>
/* Private typedef -----------------------*/
/* Private define ------------------------*/
/* Private macro -------------------------*/
/* Private variables ---------------------*/
/* Private function prototypes -----------*/
/* Exported functions --------------------*/
/**
    @brief  Bla, bla
    @param  peters  Bla, bla
    @retval The number of pepes
*/
uint8_t template_init(uint16_t peters)
{
    return (uint8_t)(peters + 1u);
}
/* Private functions ---------------------*/
/* End of file ****************************/
```

### Own header first

The compiler checks that the code matches the contract, and that the .h compiles on its own

### Private sections

What is used only inside this module and is not published in the .h

### Function comment

**`@brief`**, **`@param`**, **`@retval`** before every function

### How to use it

Copy both files, rename them (**`area.h`**, **`area.c`**), replace **`template`** and fill in the comments

# Splitting into modules

- Let's build the project
  - First, we **compile** the modules. In this case, *area.c*

**C:\...\first_project> gcc -c area.c -std=c99 -Wall -Wextra -Wpedantic**

  - Note the *–c* option (compile)
  - It produces an **object** file named *area.o* containing machine code for the target architecture and OS

```
H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex02>dir
 Volume in drive H is VBOX_home_aperles
 Volume Serial Number is 0001-0302

 Directory of H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex02

27/10/2026  12:10    <DIR>          .
27/10/2026  11:34                96 area.h
27/10/2026  11:20    <DIR>          ..
27/10/2026  12:04               834 area.o
27/10/2026  11:26               246 main.c
27/10/2026  11:33               146 area.c
               4 File(s)          9,514 bytes
               2 Dir(s)  726,547,722,240 bytes free

H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex02>
```

# Splitting into modules

- Let's build the project (contn'd)
  - Now we compile *main.c* and link it to the rest of the modules. In this case *area.o*

**C:\...\first_project> gcc main.c area.o -o hello_circle2 -std=c99 -Wall -Wextra -Wpedantic**

  - The *hello_circle2.exe* app should be available for execution now

```
H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex02>dir
 Volume in drive H is VBOX_home_aperles
 Volume Serial Number is 0001-0302

 Directory of H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex02

27/10/2026  12:16            40,787 hello_circle2.exe
27/10/2026  12:16    <DIR>          .
27/10/2026  11:34                96 area.h
27/10/2026  11:20    <DIR>          ..
27/10/2026  12:04               834 area.o
27/10/2026  11:26               246 main.c
27/10/2026  11:33               146 area.c
               5 File(s)         50,301 bytes
               2 Dir(s)  726,547,599,360 bytes free

H:\dades\docencia\cursos impartits\curs C-ESIGELEC\clases\3 Toolchain\code\toolchain_ex02>
```

# Splitting into modules

- Why compile separately?
  - The whole project can also be built in one step

C:\...\first_project> **gcc main.c area.c -o hello_circle2 -std=c99 -Wall -Wextra -pedantic**

  - So why bother with **-c** and *.o* files?

*Think of a project with 500 .c files… and you change one line in one of them*

- Only the modified module is recompiled; then everything is relinked
  - Build tools automate this: *make*, *CMake*, IDEs…

# Which stage failed?

- Two broken builds. For each one: **compiler** or **linker**? How do you know?

1. *#include "area.h"* was removed from *main.c*

```
> gcc -c main.c -std=c99 -Wall -Wextra -pedantic
main.c:10:48: warning: implicit declaration of function 'area_circle'
main.c:10:42: warning: format '%f' expects argument of type 'double',
              but argument 2 has type 'int'
```

2. *area.o* was forgotten when linking

```
> gcc main.c -o hello_circle2 -std=c99 -Wall -Wextra -pedantic
ld: ...main.o: in function `main':
main.c:(.text+0x54): undefined reference to `area_circle'
collect2: error: ld returned 1 exit status
```

Bonus: why does the compiler think *area_circle()* returns *int*?

# Automating the build: make

- *make* builds a project following the **rules** written in a text file named *Makefile*

```
CFLAGS = -std=c99 -Wall -Wextra -pedantic

hello_circle2: main.o area.o
    gcc main.o area.o -o hello_circle2

main.o: main.c area.h
    gcc -c main.c $(CFLAGS)

area.o: area.c area.h
    gcc -c area.c $(CFLAGS)
```

**target**
the file to build

**dependencies**
the files it is built from

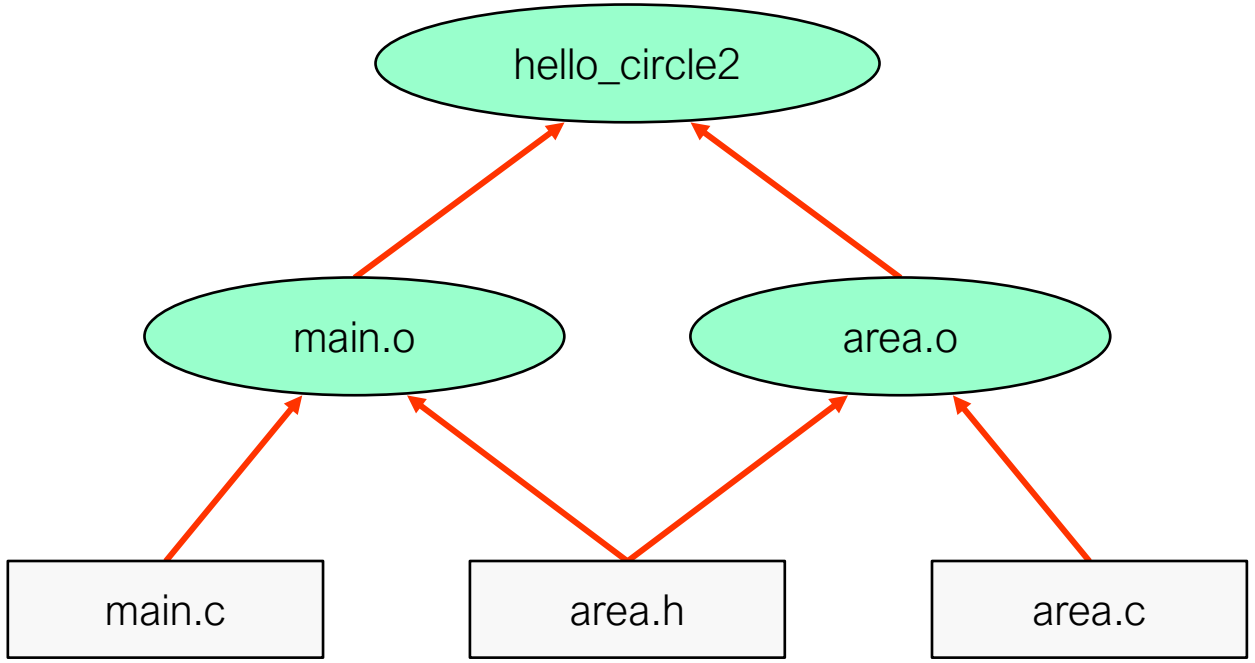
**recipe**
the command; its line must start with a TAB

**$(CFLAGS)**
a variable, replaced by its value

Run *make*: it builds the **first target** (hello_circle2), building first the targets it depends on
Each rule = *what* to build + *from what* + *how*

# How make decides what to rebuild

- make compares **file dates**: a target is rebuilt only if one of its dependencies is newer than it




*arrow = "is used to build"*

You edit *area.c* and run *make*:

```
> make
gcc -c area.c -std=c99 -Wall ...
gcc main.o area.o -o hello_circle2
```

*main.c* is not recompiled: *main.o* is still newer than its dependencies

## Questions

What happens if you run *make* again?
And if you edit *area.h*?

# make in the course template

- You will not write Makefiles in this course: the project template brings a **generic** one

```
my_lab/
├── Makefile
├── src/       your .c files
├── include/   your .h files
├── build/     made by make
└── .vscode/   VS Code setup
```

### Generic rules

Every .c in src/ is compiled and linked. Add a module: just drop its files in place

### Header dependencies

gcc writes them for make: change a .h and the right files are rebuilt
`options -MMD -MP`

### Do not edit

the Makefile or the .vscode/ files
Never use ▷ *Run C/C++ File*: it builds only the active file and ignores the Makefile

### From the terminal

**`mingw32-make`** build
**`mingw32-make run`** build + run
**`mingw32-make clean`** delete build/

### From VS Code

Ctrl+Shift+B runs make
F5 runs make, then the debugger

***Bigger projects:*** *CMake generates the build files for you (e.g. ESP-IDF, Zephyr)*

# Comparing C and Java

| | Java | C |
|---|---|---|
| Build output | bytecode (.class) | native machine code |
| Linking | at run time (class loader) | at build time (linker) |
| import / #include | names; no text copied | text literally pasted |
| Headers | none | .h = the module's contract |
| Portability | compile once, run on any JVM | rebuild for each target |

*In Java you never needed a linker… or did you?*

ESIGELEC

ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA