PROGRAMMAZIONE PROCEDURALE

A.A. 2023/2024

MODULARIZATION

MODULAR PROGRAMMING

- Modularization is a method to organize large programs in smaller parts, i.e. the modules. Every module has a well defined interface toward client modules that specify how "services" provided by this module are made available.

gcc -o mainfile mainfile.c

mainfile.c

```
#include <stdio.h>

void myPrintHello(void) {
  printf("Hello!\n");
}

int main() {
  myPrintHello();
  return(0);
}
```

EXAMPLE

hellolib.h

```
void myPrintHello();
```

mainfile.c

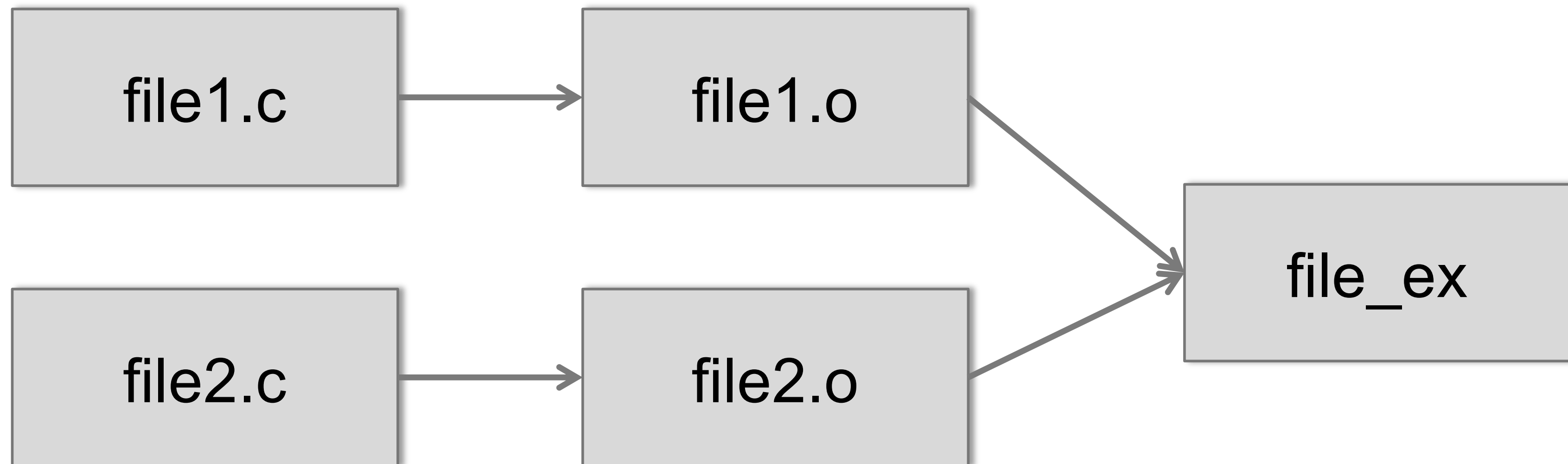
```
#include “hellolib.h”

int main() {
  // call a function in another file
  myPrintHello();

  return(0);
}
```

hellolib.c

```
#include <stdio.h>
#include “hellolib.h”

void myPrintHello(void) {

  printf("Hello!\n");

  return;
}
```

```
gcc -o mainfile mainfile.c hellolib.c
```

COMPILER AND LINKER



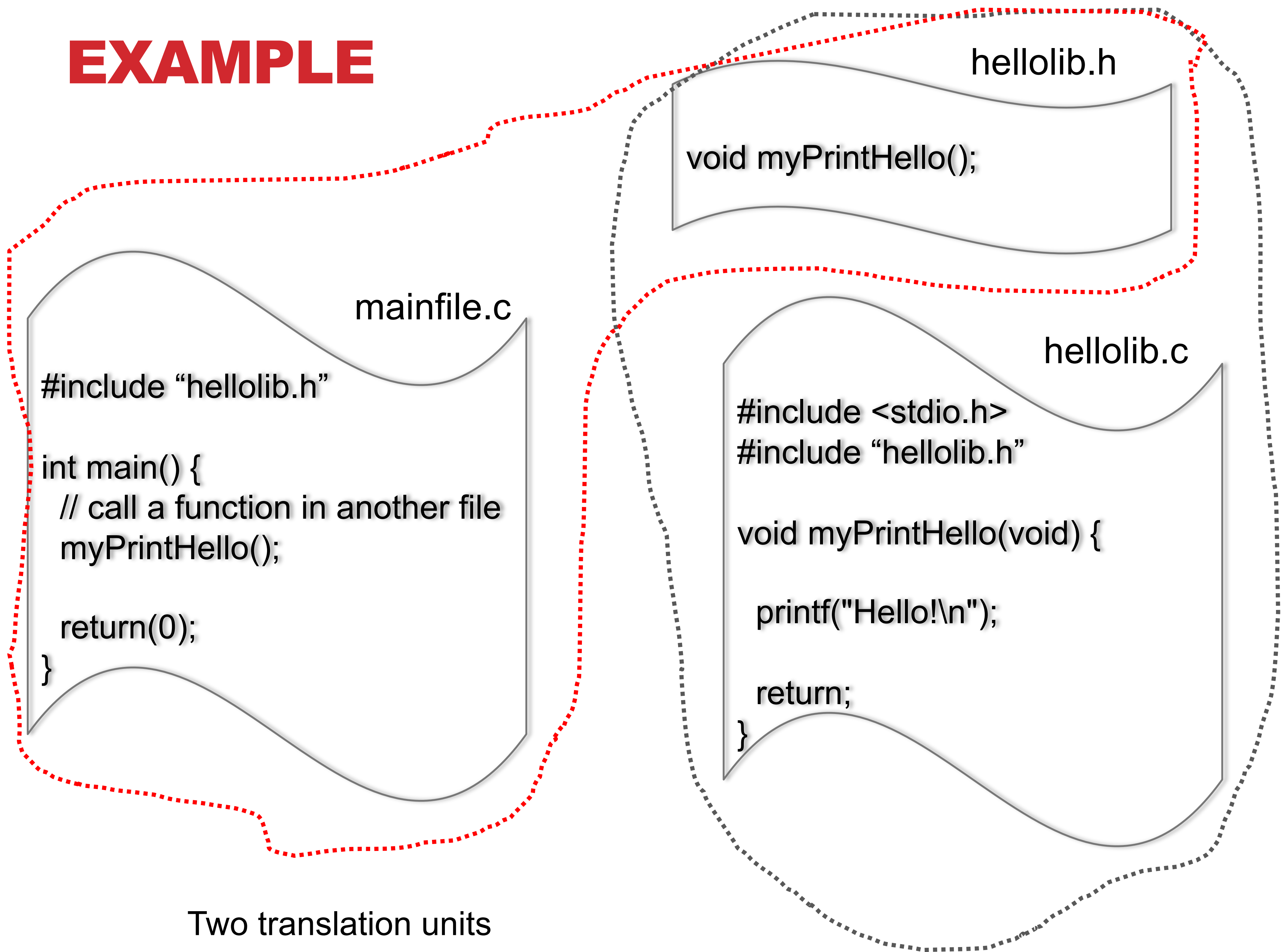
gcc -o file_ex file1.c file2.c

MODULAR PROGRAMMING

- Modularization has several benefits, especially on large and complex programs:
  - ✓modules can be re-used in several projects;
  - ✓changing the implementation details of a modules does not require to modify the clients using them as far as the interface does not change;
  - ✓faster re-compilation, as only the modules that have been modified are actually re-compiled;
  - ✓self-documenting, as the interface specifies all that we need to know to use the module;
  - ✓easier debugging, as modules dependencies are clearly specified and every module can be tested separately;

HOW THE C COMPILER WORKS

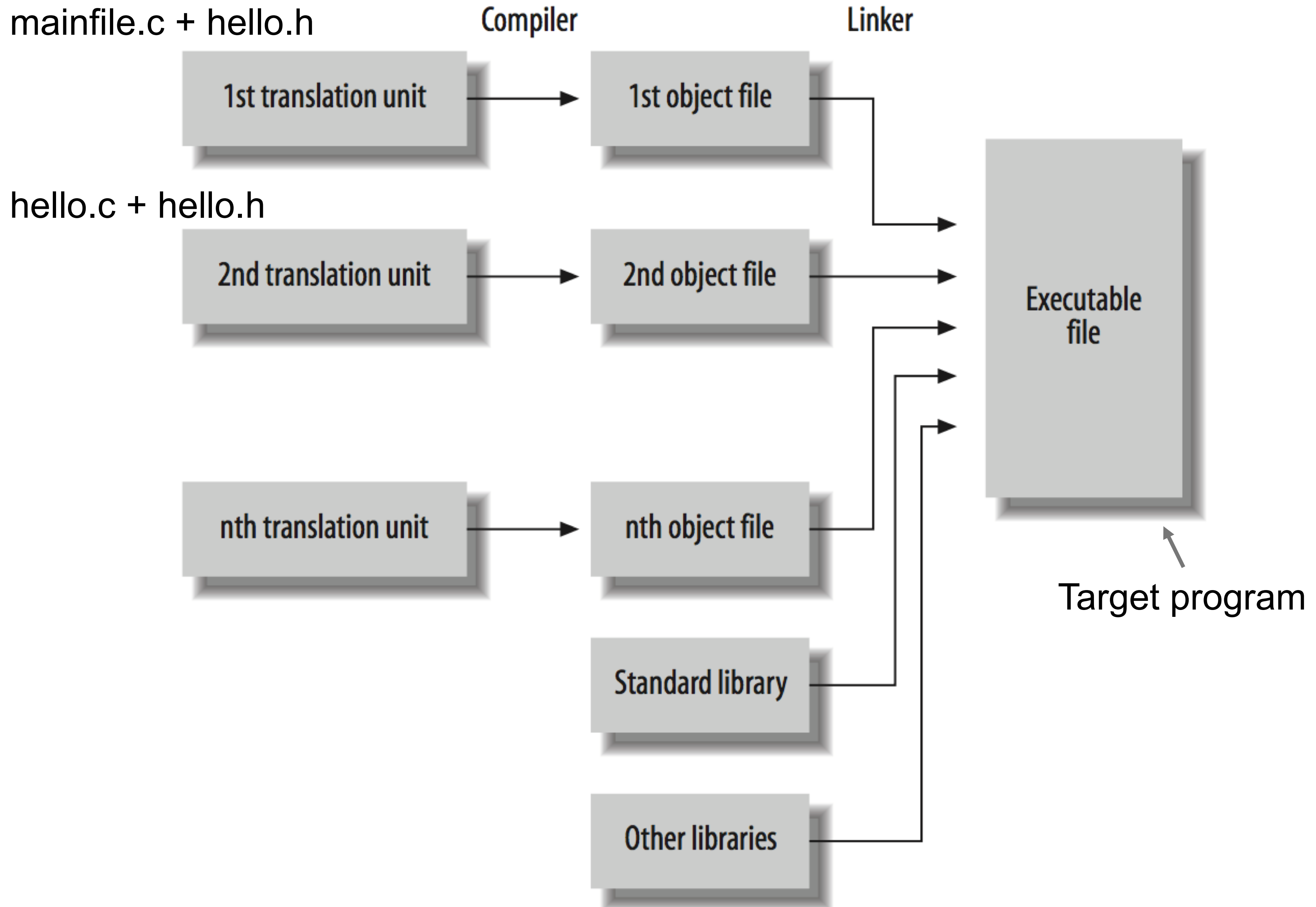
- The compiler operates on a *translation unit* consisting of
  - ✓a source file and
  - ✓all the header files referenced by #include directives.
- If the compiler finds no errors in the translation unit, it generates an *object file* containing the corresponding machine code; usually identified by the filename suffix ***.o*** or ***.obj***
- Object files are also called *modules*. A library, such as the C standard library, contains compiled, rapidly accessible modules of the standard functions.
- The compiler translates each translation unit of a C program—that is, each source file with any header files it includes—into a separate object file.


EXAMPLE
hellolib.h
void myPrintHello();
mainfile.c
#include “hellolib.h”
int main() {
// call a function in another file
myPrintHello();
return(0);
}
hellolib.c
#include <stdio.h>
#include “hellolib.h”
void myPrintHello(void) {
printf("Hello!\n");
return;
}
Two translation units

HOW THE LINKER WORKS

- The compiler then invokes the *linker*, which combines the object files, and any library functions used, in an *executable file*.
- The executable file also contains any information that the target operating system needs to load and start it.

HI, I'M THE LINKER



EXAMPLE

hellolib.h

```
void myPrintHello(void);
```

mainfile.c

```
#include "hellolib.h"

int main() {
  // call a function in another file
  myPrintHello();

  return(0);
}
```

hellolib.c

```
#include <stdio.h>
#include "hellolib.h"

void myPrintHello(void) {

  printf("Hello!\n");

  return;
}
```

mainfile.o

Unit 1

gcc –c mainfile.c

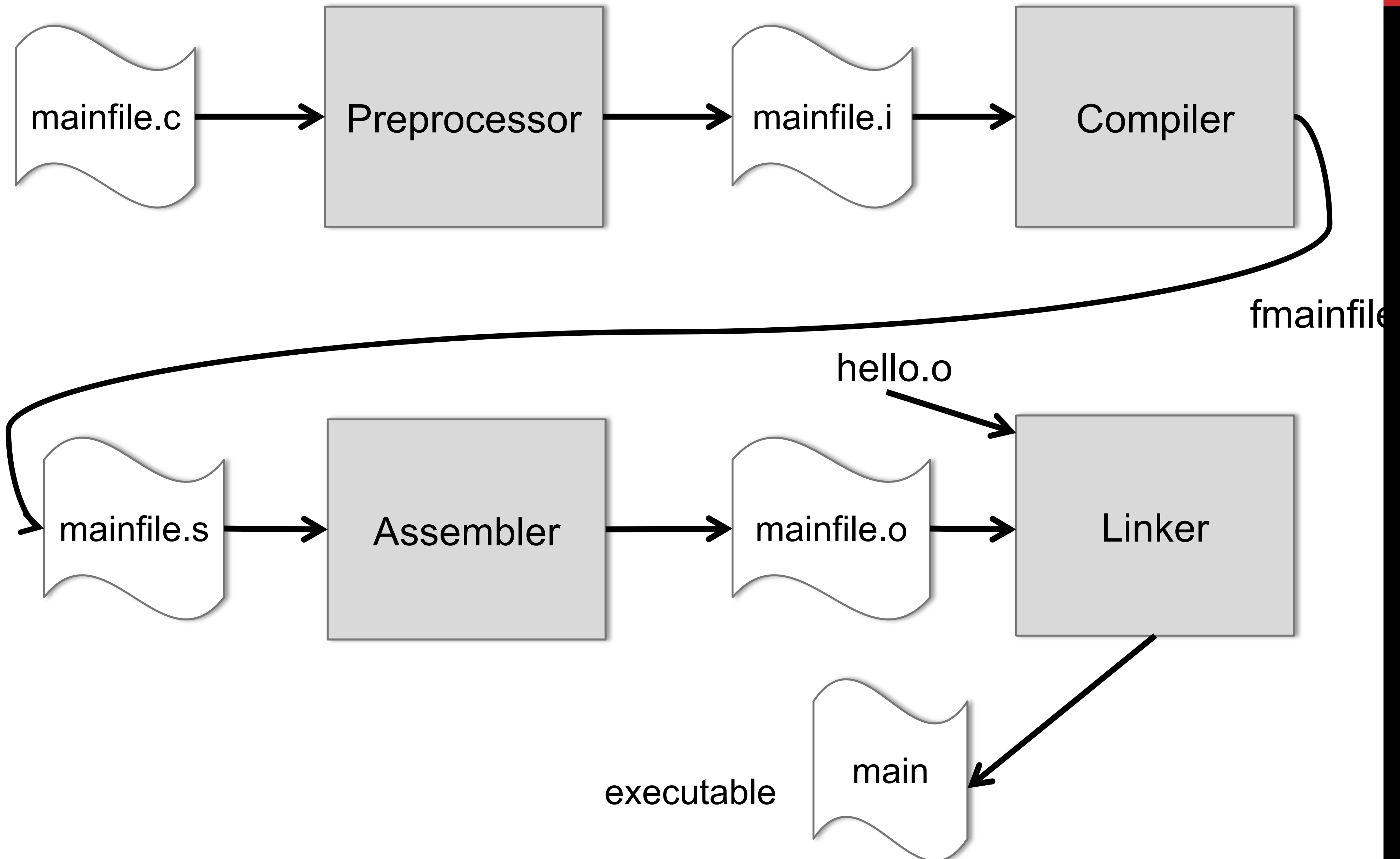
hellolib.o

Unit 2

gcc –c hellolib.c

gcc –o main mainfile.o hellolib.o

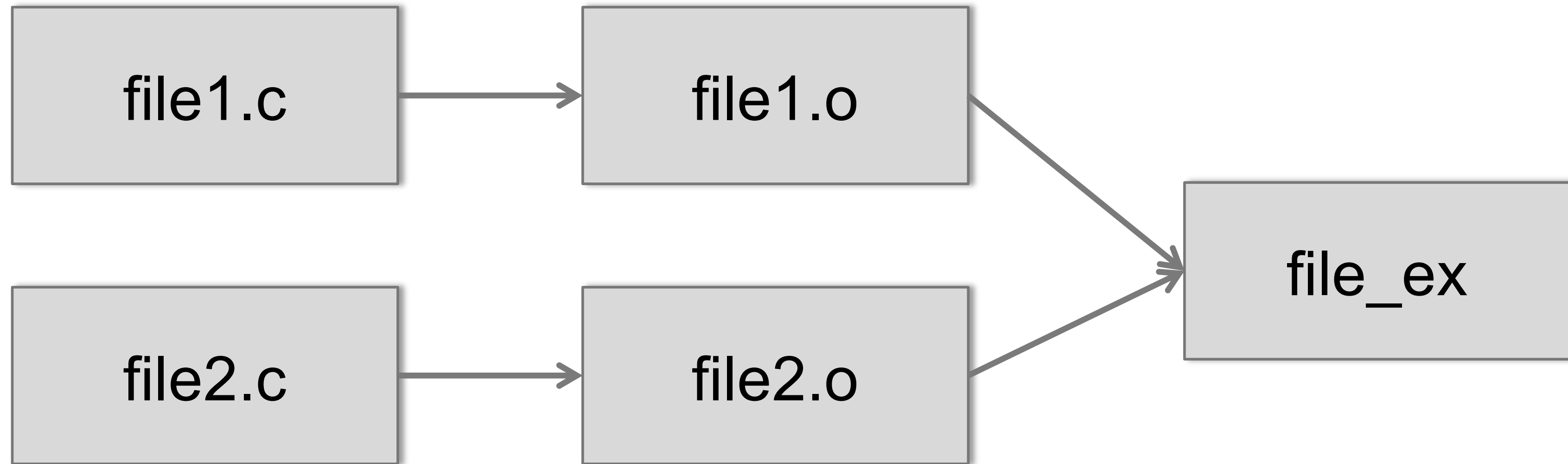
STEPS



COMPILING SEPARATELY

- Note that
  - ✓gcc -o mainfile mainfile.c hellolib.c
- Is different from compiling
  - ✓gcc –c mainfile.c
  - ✓gcc –c hellolib.c
- In this second case you are compiling the two files separately: you do not need hellolib.c to do
  - ✓gcc –c mainfile.c
- And viceversa
- You can start writing one before the other one

COMPILER AND LINKER



gcc -c file1.c

gcc -c file2.c

gcc -o file_ex file1.o file2.o

MODULAR PROGRAMMING

- When a program consists of several source files, you need to declare the same functions and global variables, and define the same macros and constants, in many of the files.
- The header (.h) contains only **macro defintion, types definition, and function declarations** that client program are allowed to see and to use.
- All these things, which a programmer wants to “offer” to other files, need to be included in the .h
- Every other private item internal to the module must stay inside the code file. We will now describe in detail the header and the code file.

EXAMPLE

hello.h

```
void myPrintHello();

enum color {black, white};

const double p= 3.14;
```

mainfile.c

```
#include “hello.h”

int main() {
  // call a function in another file
  myPrintHello();

  return(0);
}
```

hello.c

```
#include <stdio.h>
#include “hello.h”

void myPrintHello(void) {

  printf("Hello %f!\n”, p);

  return;
}
```

```
gcc -o mainfile mainfile.c hello.c
```

IMPORTANT

- In hellolib.c **INCLUDE** hello.h
- Including its own header, the compiler grabs all the constants, types and variables it requires.
- Another useful effect of including the header is that prototypes are checked against the actual functions, so that for example if you forgot some argument in the prototype, or if you changed the code missing to update the header, then the compiler will detect the mismatch with a proper error message.

LIBRARY

- A library works exactly in this way

arithLib.h

```
double sum (double, double );
double min (double, double );
double mul (double, double );
double div (double, double );

const double p= 3.14;
```

arithLib.c

```
#include "arithLib.h"

double sum (double p1, double p2) {

    return (p1+ p2);
}

// ….
```

mainfile.c

```
#include "arithLib.h"

int  main() {
    double a= 0.0, b= 0.0;
    scanf("%f", &a);
    scanf("%f," &b);
    int result = sum(a, b);

    // Other statements
}
```

STORAGE SPECIFIERS

STORAGE CLASS SPECIFIERS

- A ***storage class specifier*** in a declaration modifies the **linkage** of the identifier declared, and the **storage duration** of the corresponding objects.
  - ✓We have seen storage duration in previous slides.
- Storage specifier: auto, register, **static**, <u>**extern**</u>.
- A frequent source of confusion in regard to C is the fact that **linkage, which is a property of identifiers**, and **storage duration, which is a property of objects**, are both influenced in declarations by the same set of keywords—the storage class specifiers.
- <u>Remember: objects have storage duration, not linkage; and identifiers have linkage, not storage duration.</u>

STORAGE CLASS

- No more than one storage class specifier may appear in a declaration.
  - ✓No -> static extern int a;

- Function identifiers may be accompanied only by the storage class specifier **extern** or **static**.
- Function parameters may take only the storage class specifier **register**.
- Variable identifiers may have **auto**, **register**, **static**, **extern**.

LINKAGE

- The linkage of an identifier defines if that identifier can be used in other translation units or not
- A sort of scope among different files
- A function identifier can have
  - ✓**Internal** or **external linkage**
- A variable identifier can have
  - ✓**Internal**, **external**, or **no linkage**

STATIC

- Static: a function identifier declared with the specifier **static** has **internal linkage**.
- **Such an identifier cannot be used in another translation unit to access the function.**
- An object identifier declared with static has either no 1) linkage or 2) internal linkage, depending on whether the object’s definition is 1) inside a function or 2) outside all functions.
- Objects declared with static always have static storage duration. Thus the specifier static allows you to define local objects—that is, objects with block scope—that have static storage duration.

EXTERN

- Extern: function and object identifiers declared with the extern specifier have external linkage:
  - ✓You can use them anywhere in the entire program.

- The compiler treats function declarations without a storage class specifier as if they included the specifier **extern**. They have external linkage.
- Similarly, any variable identifier that you declare outside all functions and without a storage class specifier have external linkage. Without extern, the identifiers of global variables have external linkage as well.

NO LINKAGE

- Variable identifiers that are local to functions, without **extern** specifier, that is no specifier or **static**, have no linkage
- Also parameters of functions have no linkage: they are local to their function

```
static int c;

int fun (int a, int b) {
    int res= a + b;
    extern int c
    static int d= res + c;
    return res;
}
```

a, b, res,d have **no linkage**

c has the linkage of the name it refers to, in this case, **internal linkage**

EXAMPLE (LINKAGE)

Code	Linkage
`int func1( void );`	func1 external
`int a;`	a external
`extern int b;`	b external
`static int c;`	c internal
`static void func2( int d ) {`	func2 internal, d no linkage
`extern int a;`	external, this *a* is the same as above
`int b = 2;`	no linkage
`static int e;`	no linkage
`extern int c;`	c is the same as the c above: internal
`/* ... */`	
`}`	
`void fun3 (void) {`	external
`/* commands */`	
`}`	

DECLARATION AND DEFINITION OF VARIABLES

DECLARATION OF FUNCTIONS

- A function declaration is a definition if it contains the function block.

```
int iMax( int a, int b )                 // This is the function's definition. a and
                                         // b are defined
{
    return ( a >= b ? a : b );
}
```

DECLARATION OF VARIABLES

- **An object declaration is a definition if it allocates storage for the object.**
- **With extern specifier, a variable is declared**
- **Declarations that include initializers are always definitions.**
- Furthermore, all declarations within function blocks are definitions unless they contain the storage class specifier extern.

```
extern int a = 10;        // Definition of a.
extern double b;          // Declaration of b, which needs to be
                          // defined elsewhere in the program.

void func(int a) {        // Definition of a
  extern char c;          // Declaration of c, not a definition.
  static short d;         // Definition of d.
  float e;                // Definition of e.
}
```

BACK TO LINKAGE WITH EXAMPLES

EXAMPLE 1

main.c

```
static int count=5;
void write_extern();

 int main() {
     write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

count (defined) has internal linkage in main.c: visible only in main.c

write_extern (declared) has external linkage in main.c: is a declaration of function. Its definition is elsewhere

count (declared) has external linkage in write.c

write_extern (defined) has external linkage in main.c: it is a definition of function: it is visible in other translation units

EXAMPLE 1

main.c

```
static int count=5;
void write_extern();

 int main() {
     write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

```
gcc -o main write.c main.c
```

```
Undefined symbols for architecture x86_64:
  "_count", referenced from:
      _write_extern in write-e81d6a.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

EXAMPLE 2

main.c

```
int count= 5;
void write_extern();

 int main() {
     write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

static void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

count (defined) has external linkage in main.c: visible also in other translation units

write_extern (declared) has external linkage in main.c: is a declaration of function. Its definition is elsewhere

count (declared) has external linkage in write.c

write_extern (defined) has internal linkage in write.c: it is not visible elsewhere

EXAMPLE 2

main.c

```
int count= 5;
void write_extern();

 int main() {
     write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

static void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

```
gcc -o main write.c main.c

MacBook-Francesco:ProgrammI francescosantini$ gcc -o main main.c write.c
Undefined symbols for architecture x86_64:
  "_write_extern", referenced from:
      _main in main-a3af3a.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

EXAMPLE 3

main.c

```
int count=5;
void write_extern();

 int main() {
     write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

extern void write_extern(void)
{
  printf("count is %i\n", count);
}
```

With or without it is the same

gcc –c main.c

gcc –c write.c

gcc –o main main.o write.o

OK!

EXAMPLE 4

main.c

```
int count=5;
void write_extern();

int main() {
    write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

```
gcc –o main main.c

Undefined symbols for architecture x86_64:
  "_write_extern", referenced from:
      _main in main-95ef15.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

EXAMPLE 5

main.c

```
int count=5;
void write_extern();

 int main() {
     write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

gcc –o write write.c

```
Undefined symbols for architecture x86_64:
  "_count", referenced from:
      _write_extern in write-713128.o
  "_main", referenced from:
     implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

TENTATIVE DEFINITION

- If you declare an object outside of all functions, without an initializer and without the storage class specifier extern, the declaration is a *tentative definition*.
- A tentative definition of an identifier remains a simple declaration if the translation unit contains another definition for the same identifier.
- If not, then the compiler behaves as if the tentative definition had included an initializer with the value zero, making it a definition.

EXAMPLE OF TENTATIVE DEFINITION

main.c

```
#include<stdio.h>

int count;
int count= 4;

int main (void) {
   printf("count is %i\n", count);
 }
```

gcc main.c

OK!: count is 4

main.c

```
#include<stdio.h>

int count= 3;
int count= 4;

int main (void) {
   printf("count is %i\n", count);
 }
```

gcc main.c

```
MacBook-Francesco:ProgrammI francescosantini$ gcc write.c
write.c:5:5: error: redefinition of 'count'
int count= 4;
    ^
write.c:4:5: note: previous definition is here
int count= 3;
    ^
1 error generated.
```

EXAMPLE

main.c

```
int count=5;
void write_extern();

 int main() {
     write_extern();
}
```

write.c

```
#include<stdio.h>

extern int count;

void write_extern(void)
 {
   printf("count is %i\n", count);
 }
```

```
gcc -o main write.c main.c
```

OK!

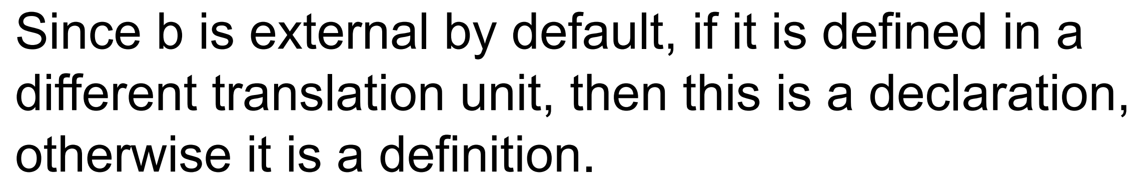
TO SUM UP

LINKAGE

- An identifier with **external** linkage represents the same function or object throughout the program. The compiler presents such identifiers to the linker, which resolves them with other occurrences in other translation units and libraries.
- An identifier with **internal** linkage represents the same object or function within a given translation unit. The identifier is not presented to the linker. As a result, you cannot use the identifier in another translation unit to refer to the same object or function.
- **No linkage**
  - ✓Function parameters
  - ✓Object identifiers that are declared within a function and without the storage class specifier extern

EXAMPLE (DEFINITION OR DECLARATION)

Code	
`int func1( void );`	declaration
`int a= 3;`	definition
`int b;`	attempt of definition / declaration
`extern int b = 1;`	definition
`static int c;`	definition
`static void func2( int d ) {`	func2 definition, d definition
`    extern int a;`	declaration
`    int b = 2;`	definition
`    static int e;`	definition
`/* ... */`	
`}`	



DECLARATION OR DEFINITION?

```
1  extern int a= 3, b;
2  static double c;
3  typedef int d;
4  d e,f;
5  int f= 3;
6
7  int fun2(int g, int* h) {
8     int i;
9     int l=5, m[e];
10    extern d e;
11    //Comandi...
12 }
```

1 **a** definito
1 **b** dichiarato
2 **c** definito
4 **e** tentativo di definizione -> definizione
4 **f** tentativo di definizione -> dichiarazione
7 **fun2** definita
7 **g** e **h** definite
8 **i** definita
9 **l** e **m** definite
10 **e** dichiarato (definito alla linea 4)

LINKAGE

```
1  extern int a= 3, b;
2  static double c;
3  typedef int d;
4  d e,f;
5  int f= 3;
6
7  int fun2(int g, int* h) {
8     int i;
9     int l=5, m[e];
10    extern d e;
11    //Comandi...
12 }
```

1 **a** external
1 **b** external
2 **c** internal
4 **e** external
4 **f** external
7 **fun2** external
7 **g** e **h** no linkage
8 **i** no linkage
9 **l** e **m** no linkage
10 **e** ha il linkage di e a linea 4

```
1 #define A 3
2 int a= A;
3 extern int b;
4 extern int compare(double r1, double
      r2);
5 static double area(double r3);
6
7 int* my_func(int c) {
8      static double e= 4.0;
9      double* f= &e;
10     register int q= 4;
11     extern int g;
12     return (int*) f;
13 }
```

2 **a** definita (linkage esterno)
3 **b** dichiarata (linkage esterno)
4 **compare** dichiarata (linkage esterno)
5 **area** dichiarata (linkage interno)
I parametri nelle dichiarazioni sono dichiarati
7 **my_func** definita (linkage esterno)
7 **c** definita (no linkage)
8 **e** definita (no linkage)
E le altre?

SU LIBRO E RIFERIMENTI

- Sezione 1.9.2
- Sezione 5.2
- Sezione 15.4
- Dichiarazione e definizione
  - ✓https://www.cprogramming.com/declare_vs_define.html
  - ✓https://stackoverflow.com/questions/1410563/what-is-the-difference-between-a-definition-and-a-declaration
- Su linkage
  - ✓https://www.geeksforgeeks.org/internal-linkage-external-linkage-c/
  - ✓https://aticleworld.com/linkage-in-c/