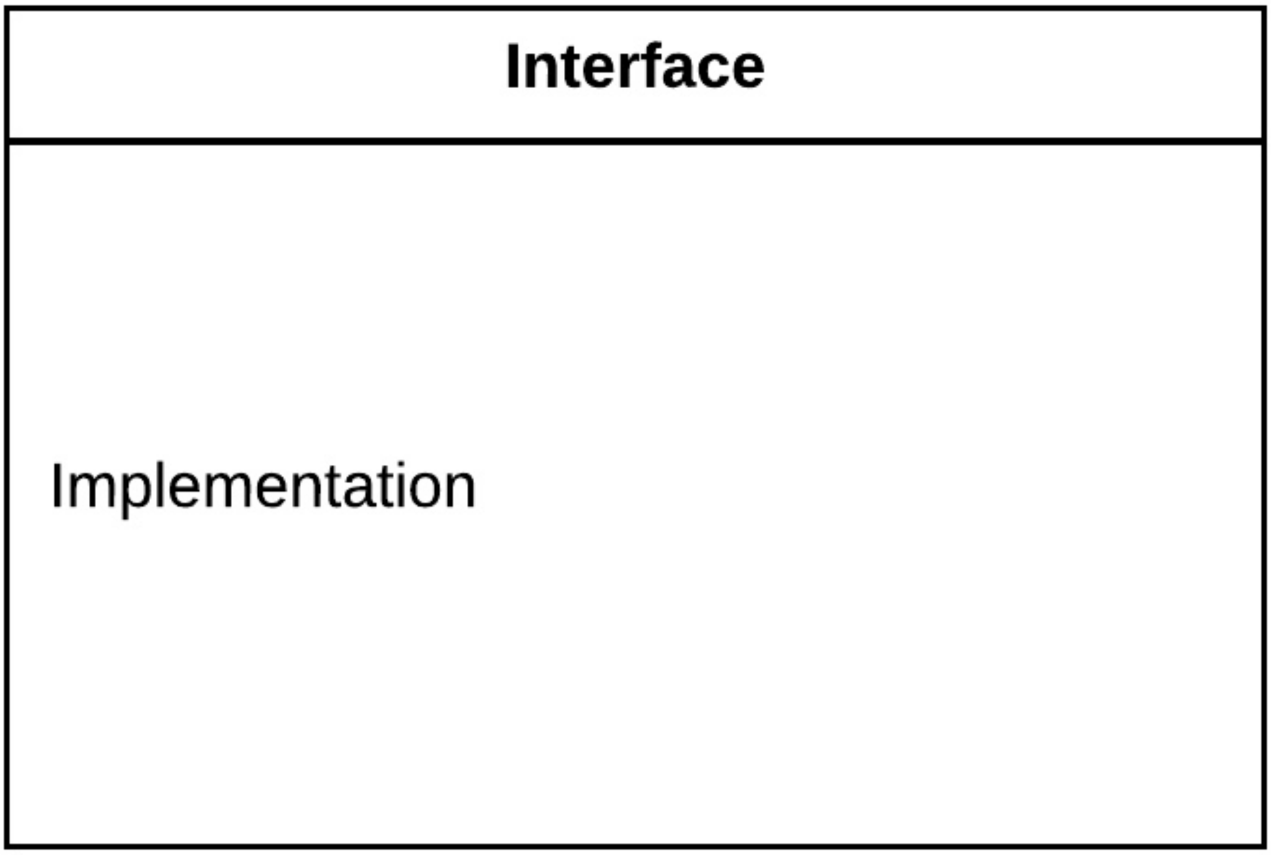
MODULARITY IN C

Module: interface + implementation.

- Interface: **what a module does**. *Declares* identifiers, types, and routines that are available to the clients.
- Implementation: **how it does it**.
- One interface, many implementations



^ Notes

^ A module comes in two parts, its interface and its implementation. The interface specifies what a module does. It declares identifiers, types, and routines that are available to the clients. An implementation specifies how a module accomplishes the purpose advertised by its interface.

^ For a given module, there is usually one interface, but there might be many implementations. Each implementation might use different algorithms and data structures, but they all must meet the specification given by the interface. This methodology helps to avoid unnecessary code duplication and helps avoid bugs: interfaces and implementations are written and debugged once, but used often. An interface specifies only those identifiers that clients may use, hiding irrelevant representation details and algorithms. This helps clients avoid dependencies on a particular implementations.

MODULARITY IN C

Coupling: dependency between the client and the server implementation

:arrow_right: It causes bugs when an implementation changes.

A well-designed and precisely specified interface reduces coupling.

MODULARITY IN C

Interface is specified by a **header file** (.*h* extension).

A client **imports** an interface with the C preprocessor `#include` directive.

The `#include` preprocessor directive causes the compiler to replace that line with the **entire text of the contents of the named file**.

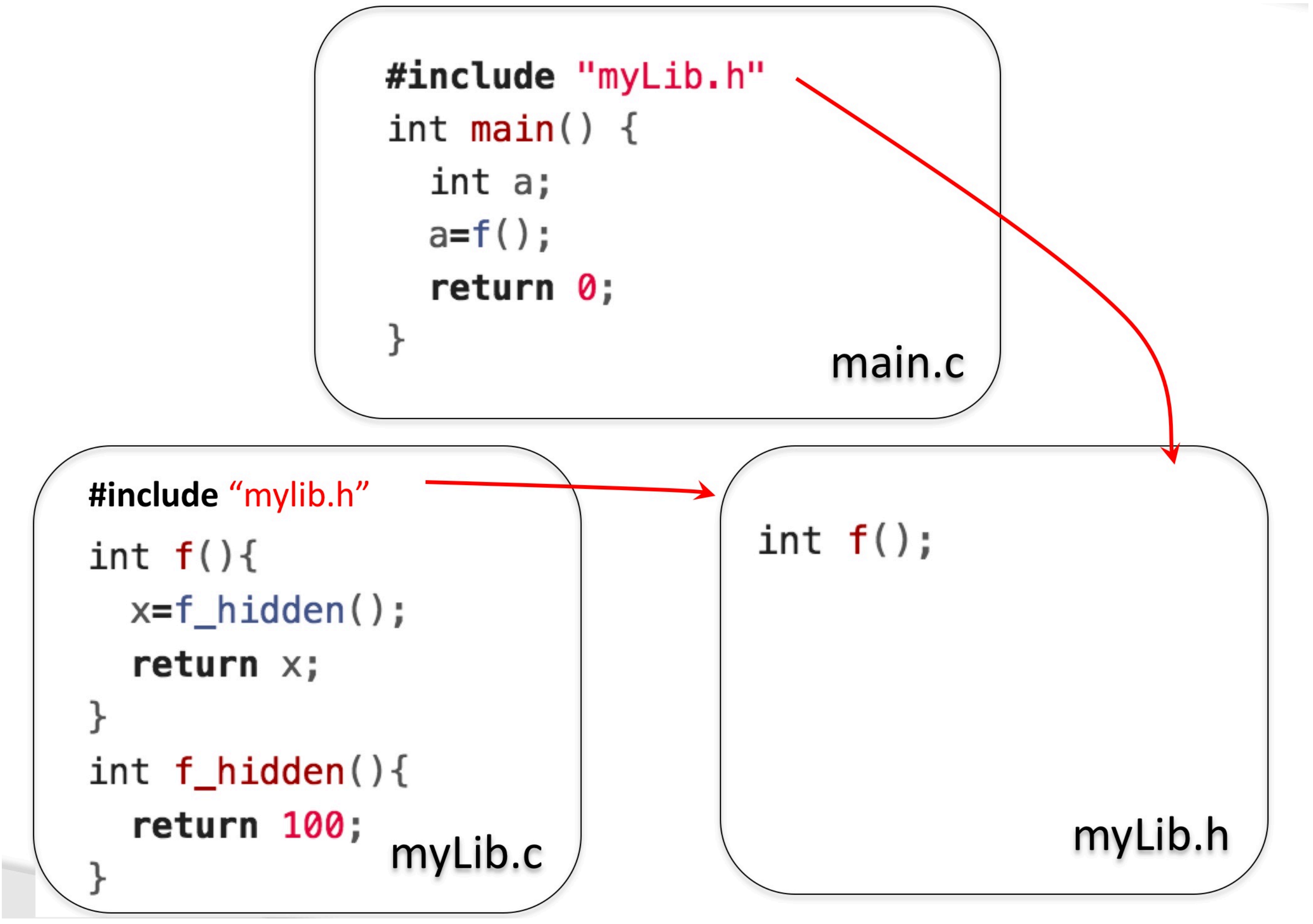
MODULARITY IN C

*Implementation includes **its** interface* (`.h` file) to ensure consistency.

Header files that **declare** functions (or external variables) should be **included** in the file that *defines* the function or variable.

i.e. `myFile.c` must include `myFile.h`

That way, the compiler can do type checking and the external declaration will always agree with the definition.



```
#include "myLib.h"
int main() {
  int a;
  a=f();
  return 0;
}
```

```
#include "mylib.h"
int f(){
  x=f_hidden();
  return x;
}
int f_hidden(){
  return 100;
}
```

```
int f();
```

who includes who?

The gold rule is: **#include .h files into .c files**

Avoid to #include .h files in other .h files.

Every header file .h should #include every other header file that it requires to compile correctly, **but no more**.

The reason is simple: if **file_1.h** includes **files_2.h** that includes **file_3.h** that includes **file_4.h**... when your mainfile.c includes file_1.h, you are including an** uncontrollable chain of header files**.

If you include .h files into .c files, you have more control over what you include

MODULARITY IN C

`#include <file>` This form is used for system header files. It searches for a file named 'file' in a standard list of system directories.

`#include "file"` This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file.

Notes

An interface is specified by a header file (.h extension). A client imports an interface with the C preprocessor `#include` directive.

An implementation is provided by one or more .c files. *Implementation includes its interface's .h file* to ensure that its definitions are consistent with the interface's declarations.

You request to use a header file in your program by **including it with the C preprocessing directive #include.**

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

Example

- Write a program that compute the **factorial** on an integer $n, n \leq 7$, using the function **myfactorial**.
- Split the program in
  - a **main**, that calls the function **myfactorial**, and
  - a *module* **mymath** that contains the function **myfactorial**.
- The module **mymath** must also provide the value $\pi = 3.14159...$, so all client modules can use this value.

(write here your solution!) https://codeboard.io/projects/136368

- Write a program that compute the **factorial** on an integer $n, n \leq 7$, using the function **myfactorial**.

```c
#include <stdio.h>
int myfactorial(int n);
int main() {
  int n=4;
  printf("%d!= %d\n",n,myfactorial(n));
  getchar();
  return 0;
}
int myfactorial(int n){
  int nMax=7;
  int f,i;
  if (n> nMax){
    printf("Too big!");
    return -1;
  }
  if (n==0) return 1;
  else {
    f=1;
    for (i=2; i<=n; i++)
    f=f*i;
    return f;
  }
}
```

A **main** that calls the function **myfactorial**

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

int main() {
  int n=4;
  printf("%d!= %d\n",n,myfactorial(n));
  getchar();
  return 0;
}
```

A *module* **mymath** that contains the function **myfactorial**.

```c
//****** mymath.h ******
int myfactorial(int n);
```

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

int myfactorial(int n){
  int nMax=7;
  int f,i;
  if (n> nMax){
    printf("Too big!");
    return -1;
  }
  if (n==0) return 1;
  else {
    f=1;
    for (i=2; i<=n; i++)
    f=f*i;
    return f;
  }
}
```

- The module **mymath** must provide also the value $\pi = 3.14159...$, so all client modules can use this value.

What's the right place? .c or .h?

```
//****** mymath.c ******
float pi=3.14159;
```

NO!

Clients module can not access the module implementation (.c).

```
//****** mymath.h ******
float pi=3.14159;
```

NO!

If several client modules import `mymath.h` , **we define the variable more than once.** [1]

Solution

Use the header file **to declare** the data but **not to define** the data [2]. The header indicates to client modules

the existence of some data or function and indicates what their type is.

```
//****** mymath.h ******
float mypi; /* declared here; defined and initialized in mymath.c */
```

```
//****** mymath.c ******
float mypi=3.14; //exported in mymath.h
```

Solution: https://codeboard.io/projects/136370

The problem of multiple inclusion

Write a program with:

- a module that exports a **data structure** designed to store the sides of a triangle.
- a module that exports **functions** that act on triangles (i.e. compute perimeter...)
- a client module (*main*) that **use** the data structure and the functions.
- sketch a graph using the relation USE.

(write here your solution)

First attempt

- Module **T_triangle** export data type **T_triangle** (using **typedef**)
- Module **mymodule** export the funcion **computePerimeter()**

```
/* T_triangle.h */

typedef struct {
    float a, b, c;
} T_triangle;
```

```
/* mymodule.h */
#include "T_triangle.h"
// Avoid to use #include in .h file.
//  Every header file .h should #include every other header file
//  that it requires to compile correctly, *but no more*.
float computePerimeter(T_triangle t);
```

```
---------------------------------------------------------------------------
```

```
/* mymodule.c */
#include "mymodule.h"
#include "T_triangle.h" //is it necessary?
float computePerimeter(T_triangle t){
    return t.a+t.b+t.c;
}
```

```
/* main.c */
#include <stdio.h>
#include "T_triangle.h"
#include "mymodule.h"
int main() {
    T_triangle t1;
    float perimeter;
    t1.a=10;
    t1.b=12;
    t1.c=15;
    perimeter=computePerimeter(t1);
    printf ("Perimeter: %0.2f\n", perimeter);
    return 0;
}
```

What's the problem with this code?

What's the problem with this code?

T_triangle has been defined several times

```
/* main.c */
#include "T_triangle.h"
#include "mymodule.h"
```

but `mymodule.h` include `T_triangle.h`, too...

#include guard (macro guard or header guard) is used to avoid the problem of **multiple inclusion**. [see the code import02.zip]

Consider this:

```
/* things.h */
#ifndef THINGS_H
#define THINGS_H
/* rest of include file */
#endif
```

Suppose this file somehow got included several times.

The first time the preprocessor encounters this include file, `THINGS_H` is undefined, so the program proceeds to define `THINGS_H` and to process the rest of the file.

The next time the preprocessor encounters this file, `THINGS_H` is defined, so the preprocessor skips the rest of the file.

Solution: https://codeboard.io/projects/136374

MODULARITY IN C

- Do not define variables in a header file - you can declare (many times) a variable, but you must define only one.
- Do not reveal implementation in a header file
- *Always* use *include guards* in .h files
- Do not nest header files - only .c files should include .h file.

but

- Every header file should `#include` every other header file that it requires to compile correctly, **but no more**.

MODULARITY IN C

(as we saw in the first slides:)

*Implementation includes **its** interface* (`.h` file) to ensure consistency.

Header files that **declare** functions (or external variables) should be **included** in the file that *defines* the function or variable.

i.e. `myFile.c` must include `myFile.h`

That way, the compiler can do type checking and the external declaration will always agree with the definition.

But: have a look to the following code:

```
#include <stdio.h>


void f(int x);// try to change parameters!

int main(){
  int a=10;
  f(a);// try to change parameters!
  getchar();
  return 0;
}

void f(int x){ // try to change parameters!
  printf("%d\n",x);
}
```

https://codeboard.io/projects/136375

Note

The **includes guards** are not sufficient to solve the problem of multiple inclusion in the first exercise (`float mypi=3.14`).

The **include guards** work at the precompiler step.

If files are compiled separately, the problem persists.

^ Notes

^ Defining variables in a header file is often a poor idea. Frequently it is a symptom of poor partitioning of code between files.

^ Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be #included for the header to be functional.

^ `#include` instructions should only be on .c files. Only on proven absolute necessity include other headers within a header file. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common #includes in one include file. PLEASE NOTE THAT this is only **one** of the possible good practices. It is not uncommon to find .h that includes other .h

---

See also

Flight Software Branch C Coding Standard - 582-2000-005 - Version 1.1 - 11/29/05

Recommended C Style and Coding Standards https://www.doc.ic.ac.uk/lab/cplus/cstyle.html

C Coding Standard https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html

Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2ª ed

---

1. Include guards do not solve this problem. See below. ↩
2. You can **declare** a variable multiple times, but it can be **defined** only once. ↩