#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXPAROLA 30
#define MAXRIGA 80
int main(int argc, char *argv[])
{
int freq[MAXPAROLA] ; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
char riga[MAXRIGA] ;
int i, inizio, lunghezza ;
FILE * f ;
for(i=0; i<MAXPAROLA; i++)
freq[i]=0 ;
if(argc != 2)
{
fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
exit(1);
}
f = fopen(argv[1], "rt") ;
if(f==NULL)
{
fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
exit(1);
}
while(fgets(riga, MAXRIGA, f) != NULL)

Abstract Data Types

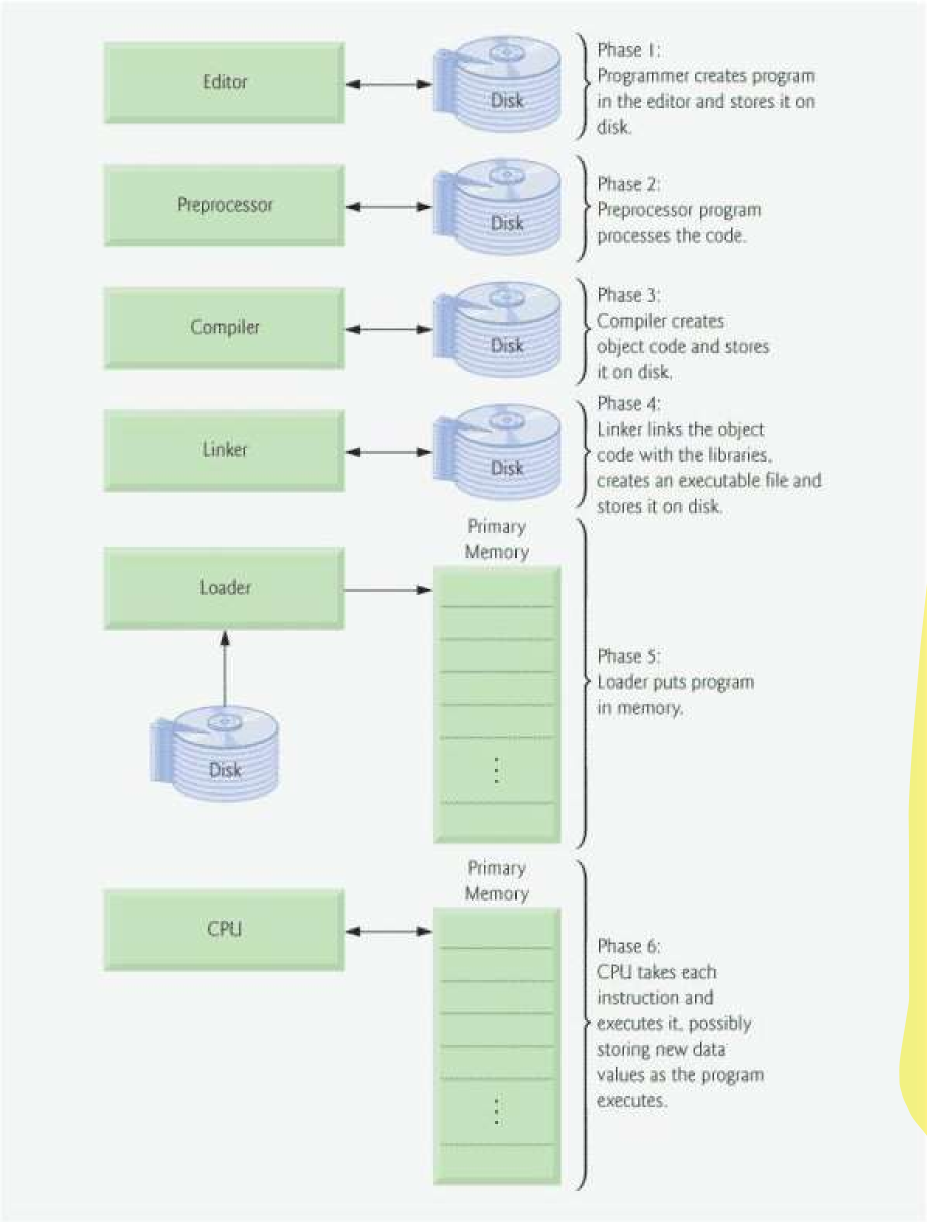
Modularity

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Software development flow



Phase 1:
Programmer creates program in the editor and stores it on disk.

Phase 2:
Preprocessor program processes the code.

Phase 3:
Compiler creates object code and stores it on disk.

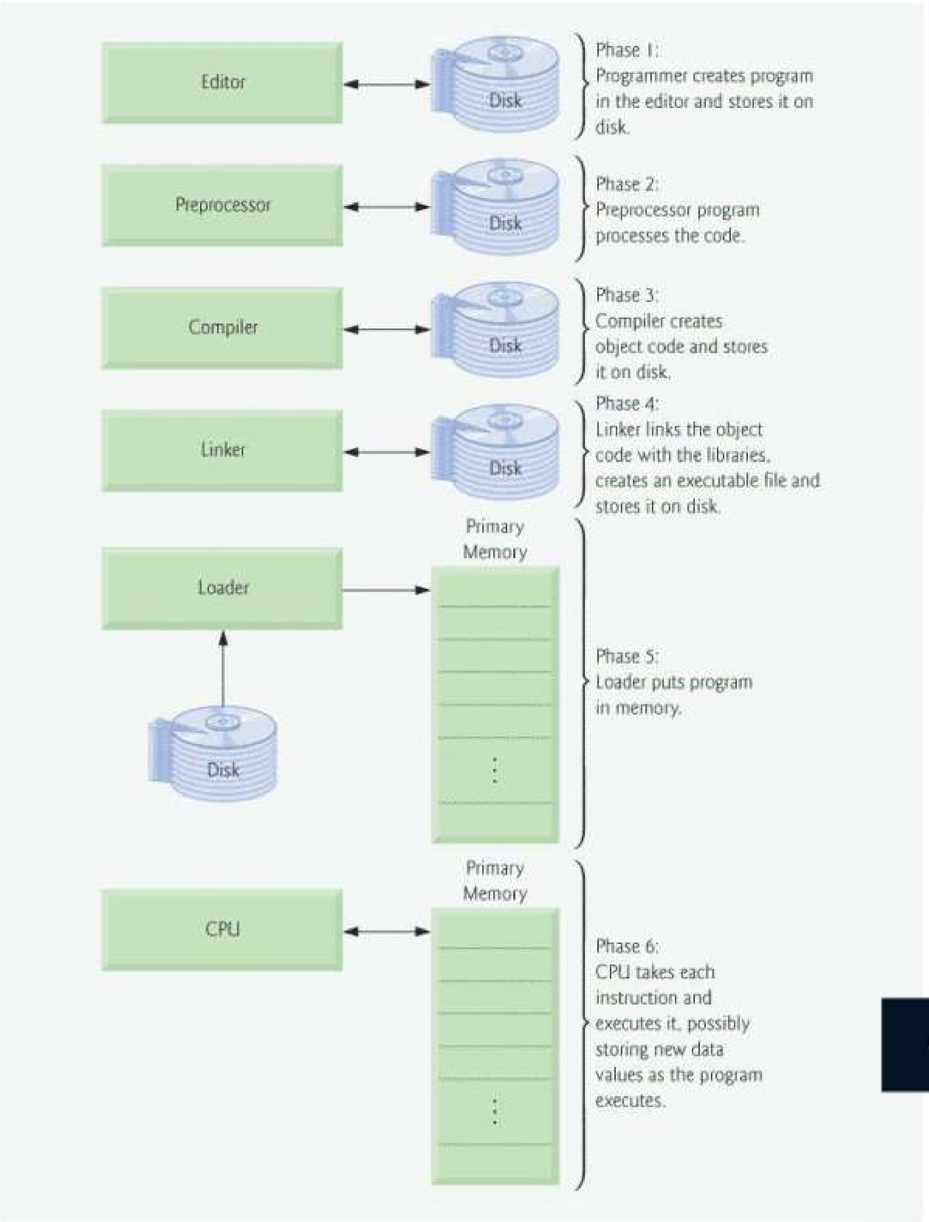
Phase 4:
Linker links the object code with the libraries, creates an executable file and stores it on disk.

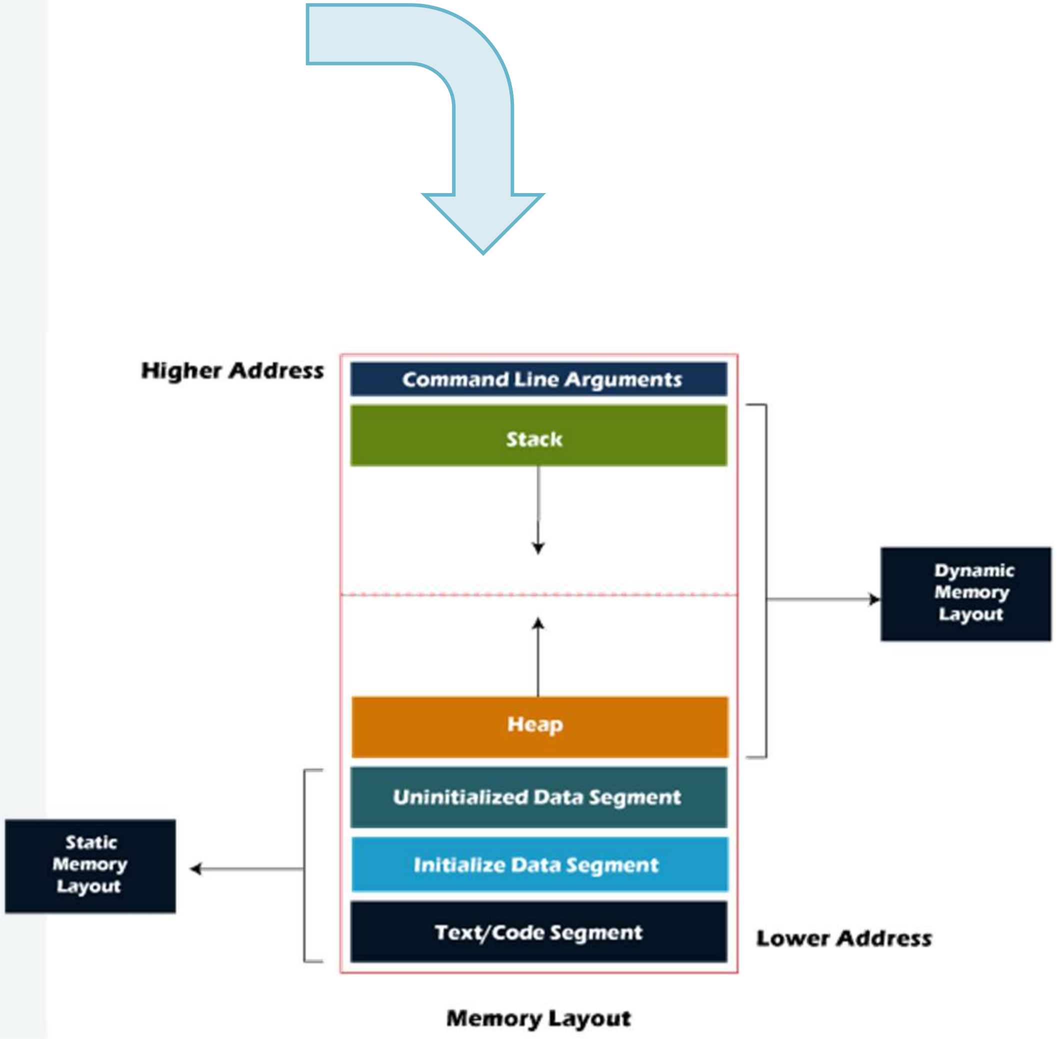
Phase 5:
Loader puts program in memory.

Phase 6:
CPU takes each instruction and executes it, possibly storing new data values as the program executes.

- Developing a program in C typically requires six phases
  - Editor
  - Pre-processor
  - Compiler
  - Linker
  - Loader
  - Execute

Software development flow


Editor
Disk
Phase 1:
Programmer creates program in the editor and stores it on disk.
Preprocessor
Disk
Phase 2:
Preprocessor program processes the code.
Compiler
Disk
Phase 3:
Compiler creates object code and stores it on disk.
Linker
Disk
Phase 4:
Linker links the object code with the libraries, creates an executable file and stores it on disk.
Loader
Primary Memory
Disk
Phase 5:
Loader puts program in memory.
CPU
Primary Memory
Phase 6:
CPU takes each instruction and executes it, possibly storing new data values as the program executes.



Small applications

- Small applications are usually included in a unique *.c file
  - It includes all required library functions
    - System libraries are declared in *.h files
    - Libraries are included with the directive
      - #include <name.h>
  - It is usually divided into a (unique) main program and several user functions
- Small applications are usually organized using two common schemes
  - All user function prototypes are inserted on top
  - Each function definition preceeds all its calls

Scheme 1

```
#include ...
#define ...
typedef ...


... function1 (...);
... function2 (...);


int main(...){...}

<type> function1 (...){...}

<type> function2 (...){...}
```

Declarations (prototypes) of all functions are inserted on top

Functions and function calls can be inserted in **any** order

Scheme 2

```
#include ...
#define ...
typedef ...







<type> function1 (...){...}

<type> function2 (...){...}

int main(...){...}
```

Declarations (prototypes) are **not** inserted

Functions and function calls **must** be inserted in a proper order

Every call must follow the relative definition

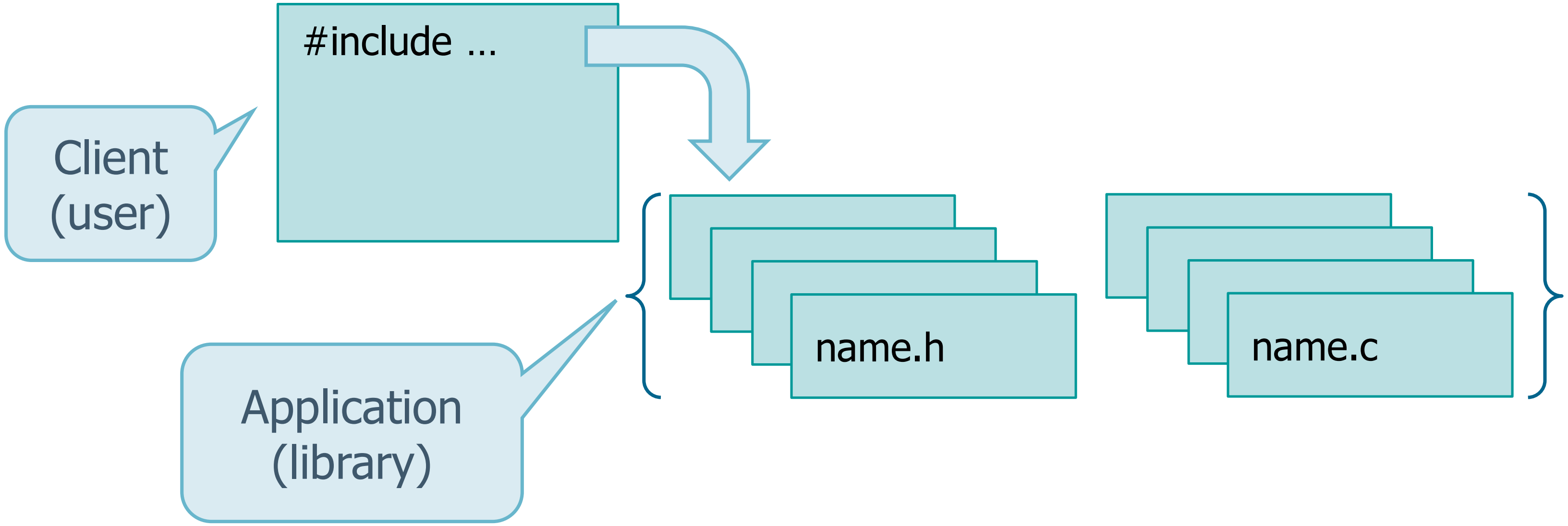
Thus, the main comes for last

Multiple-file applications

- For complex applications, source files become larger
  - They include too many functions
  - Compilation, debugging, and maintenance require long times
  - Sharing common pieces of code is practically impossible as everything is included in the same file
    - The only option is to duplicate part of the code in another file, with subsequent congruence problems

Multiple-file applications

- Large applications are written as a collection of source files
  - Header files with extension *.h
  - Source C file with extension *.c
  - These files may be stores in the same or in separate directories



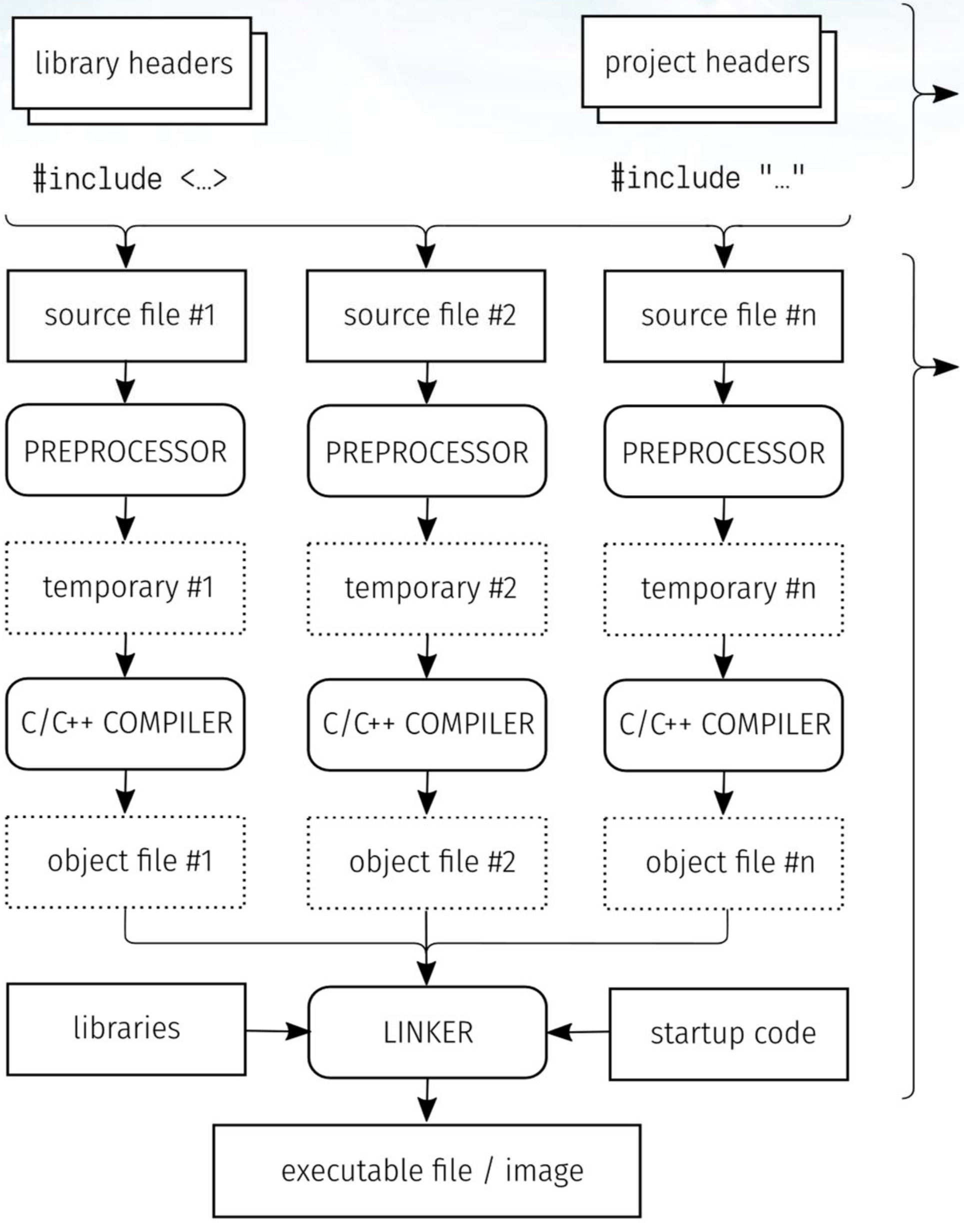
Multiple-file applications

- Usually *.c files contain
  - **Executable** instructions, i.e., C files include function definitions
    - The implementation of the main program (and all functions) must to be unique and it should appear only in one C file
  - C files are re-compiled only when needed
    - Unchanged files should not be recompiled
      - This saves time for both the programmer and the hardware platform
      - Many systems provide special utilities that recompile only the modified program files

Multiple-file applications

- Usually *.h files include
  - Functions prototypes, structure and constant definitions, etc.
    - Header files **do not include executable** instructions
  - Files
    - <name.h> include **system** libraries and system data types
    - "name.h" include **user** libraries and user data types
      - They should make public all objects the user wants to export
      - They should be included by the **client** using the library itself

Multiple-file application flow



C/C++ code that can/must be repeated in the project, due to seperate compilation.

Process is controlled by simple command line drivers like **gcc**, **g++**, **clang**, **cl**; or IDEs by way of "projects", or other build tools like **make** with "makefiles".

C/C++ compilers may produce assembler, and an assembly program produce the object file, but the process may be transparent to the programmer.

Depending on build tool used, or driver options, the object files may remain on disk, to use for future compilation.

Exporting functions

- In multi-file applications, functions must satisfy the following rules
  - Global functions
    - A module (a *.c file) who wants to export a function does not have to do anything, as all C functions are global by default
    - Each module (another *.c file) that wants to use that function has to insert its prototype, eventually (but this is optional) with the keyword **extern**
  - Local function
    - If a module (a *.c file) wants to keep a function private (i.e., it does not want to make this function global) it has to define that function as **static**

Exporting functions

- Notice that the linker (not the compiler) creates the required links between each calls and its correct function definition
- Calls and definitions must coincide, otherwise the linker complains

Exporting variables

- In multi-file applications, variables must satisfy the following rules
  - Local variables (i.e., variables defined within a function or a block) cannot be exported
  - Global variables
    - If a module (a *.c file) wants to export a global variable it does not have to do nothing, as all global C variable can be exported by default
    - Each module (another *.c file) that wants to use that variable has to insert its declaration, i.e., the keyword **extern** followed by its definition

Exporting variables

- Local variables
  - If a module (a *.c file or a function) wants to keep a variable private, i.e., it does not want to make this variable exportable, it has to define that variable as static
- Notice again that the linker (not the compiler) will create the required inks between each declaration and its corresponding definition
  - Declarations and definitions have to coincide completely, otherwise the linker complains

Example

Application with
1 C and 1 H file

Common objects

file.h

```
#include <stdio.h>

#define C1  10
#define C2 100
```

The main program

file.c

```
#include "file.h‟

int main (void> {
  int i;
  for (i=C1; i<C2; i++) {
    fprintf (stdout, "%d ", i);
   }
  return (0);
}
```

Example

Application with 3~~4~~ C and 1 H file

my.h

```
#include <stdio.h>

#define L 100
void array_read (int *, int *);
void array_write (int *, int);
```

Common objects

client.c
main.c

```
#include "my.h"

int main (void) {
  int dim;
  int vet[L];
  array_read (vet, &dim);
  array_write (vet, dim);
  return 1;
}
```

The main program
Calls 2 functions
included elsewhere

Example

read.c

```
#include "my.h"

void array_read (int *vet, int *dim){
  int i;
  printf ("Size (<%d): ", L);
  scanf ("%d", dim);
  printf ("Array:\n");
  for (i=0; i<(*dim); i++) {
    printf ("vet (%d) = ", i);
    scanf ("%d", &vet[i]);
  }
  return;
}
```

First function

Example

write.c

```
#include "my.h"

void array_write (int *vet,int dim){
  int i;
  fprintf (stdout, "Array:\n");
  for (i=0; i<dim; i++) {
    printf ("vet (%d) = %d\n",i, vet[i]);
  }
  return;
}
```

Second function

Once-Only Headers

- Each header file may include other header file
- The recursive inclusion of header file, can easily result to include the same file more than once
  - If a header file happens to be included twice, the compiler will process its contents twice which is useless and prone to errors
  - To avoid multiple inclusions C programmers use the so called "once-only header" file approach
    - Each header file is protected by multiple inclusions

Example

An example of multiple inclusion: Wrong approach

f1.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...
#define C1 10
```

f2.h

```
#include "f1.h"
...
#define C2 "abc"

...
```

f3.h

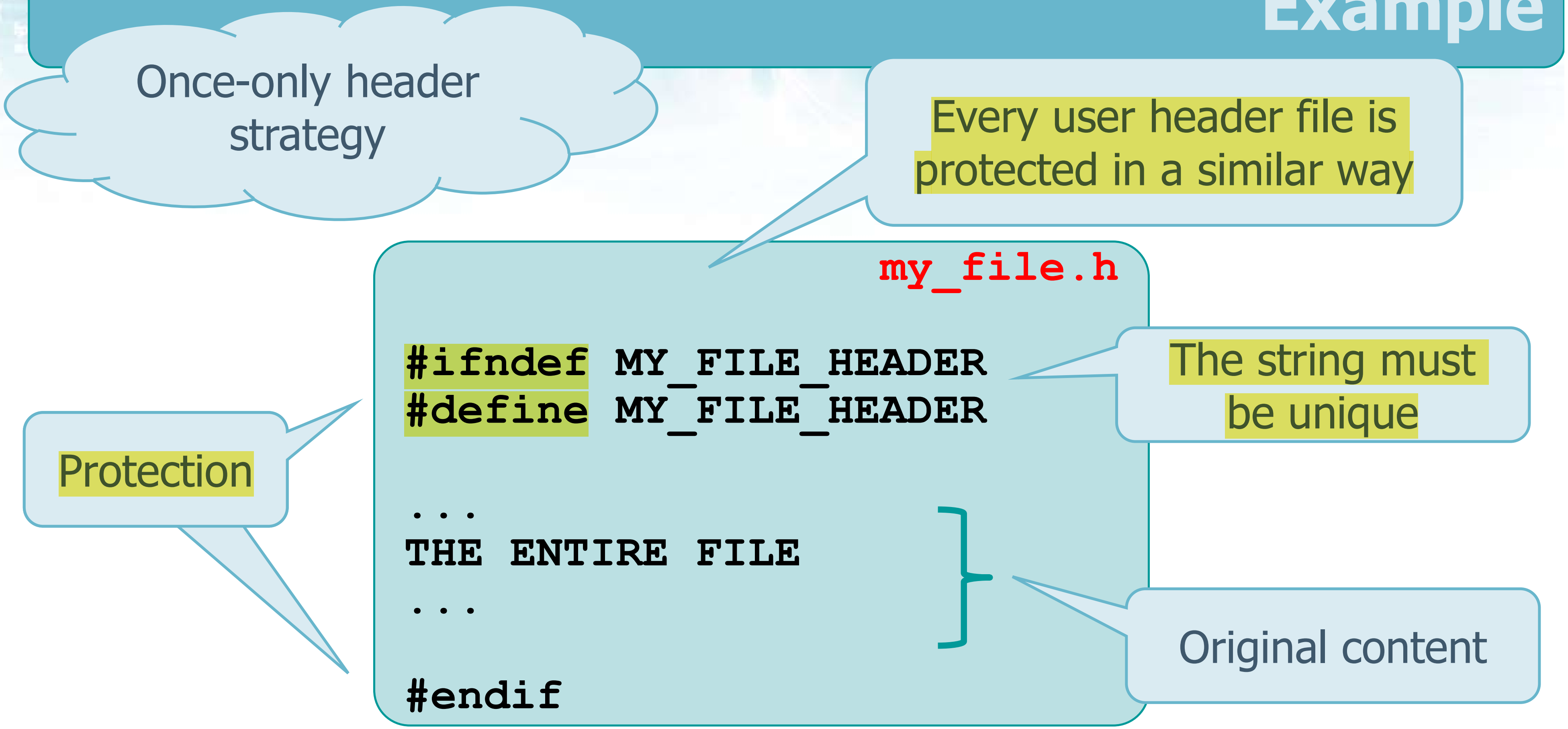
```
#include "f1.h"
#include "f2.h"
...
#define C3 12.50
...
```

Including f3, will include f1 and f2, and f2 will include f1 a second time
C1 is define more than onvce

client.c

```
#include "f3.h“
...
```

Example



Once-only header strategy

Every user header file is protected in a similar way

my_file.h

```
#ifndef MY_FILE_HEADER
#define MY_FILE_HEADER

...
THE ENTIRE FILE
...

#endif
```

Protection

The string must be unique

Original content

- If the constant **MY_FILE_HEADER**
  - Is not defined, we defined it and insert the content of the header file
  - If it is defined, we insert nothing

Example

An example of multiple inclusion: Correct approach

f1.h

```
#ifndef _F1
#define _F1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...
#define C1 10

#endif
```

f2.h

```
#ifndef _F2
#define _F2

#include "f1.h"
...
#define C2 "abc"
...

#endif
```

f3.h

```
#ifndef _F3
#define _F3

#include "f1.h"
#include "f2.h"
...
#define C3 12.50
...

#endif
```

When f1.h is inserted for the first time _F1 is not defined and the inclusion is performed. When it is included the second time _F1 is already defined and the inclusion is not done

client.c

```
#include "f3.h“
...
```