

Modular programming

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Key concepts

- Compilation versus linking
 - Compilation: generate an object file from a source file
 - Linking: generate an executable from object files
- Header file
 - Gather the data structures and function definitions used in multiple source files
 - Guarded with `#ifndef/#define/#endif`
- Libraries: brings a set of object files together
 - Static (.a): included in each executable
 - Shared (.so): shared between multiple processes

Modular programming

- Putting all the code in a single file is sometime unrealistic
 - For example, Linux is 25 millions lines of code
- Sometime, we have to aggregate codes from different entities
 - Reuse the code written in another program
- Solution: put the code in different files
 - Generate a binary from several source files

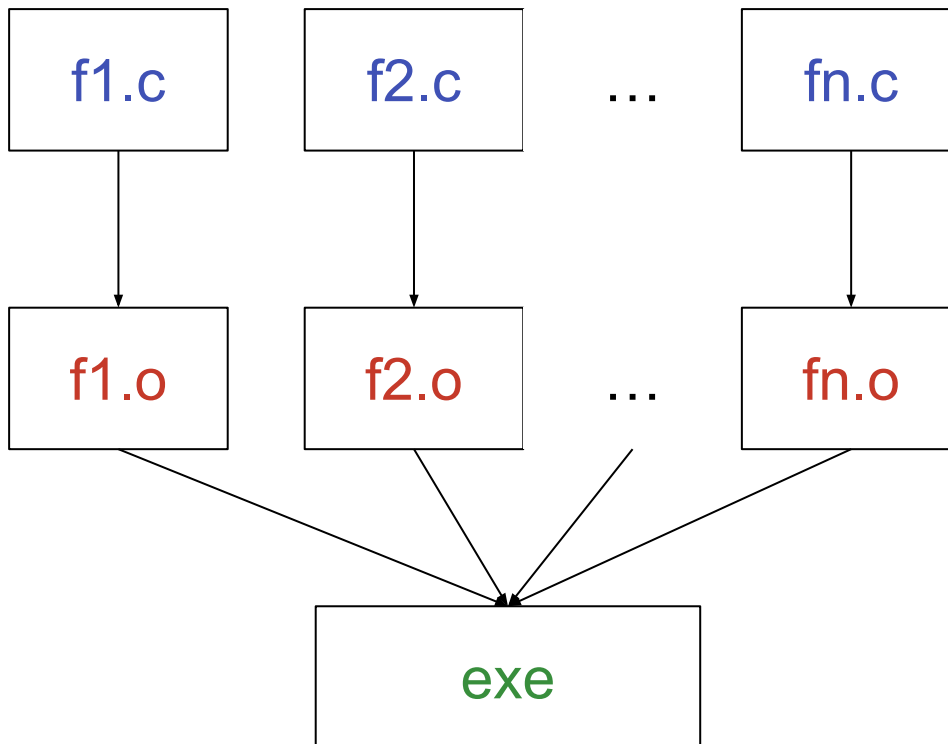
Multiple source files in a binary

- (Bad) solution:
 - Call `gcc -o binary file1.c file2.c`
- Bad solution because we have to recompile all the source files whenever a single file is modified
 - Very inefficient
- (Good) solution: split compilation in two phases
 - Compilation: translates from source to binary objects
 - Linking: aggregates several binary objects into an executable

Compilation versus linking

■ Two different commands

- **gcc -c f.c -o f.o**: compile f.c in f.o
- **gcc f.o g.o -o exe**: linke f.o and g.o into exe



source files

compilation: gcc -c

object files

linking: gcc (without -c)

executable

Problems

- What about the data structures?
 - Replicating the definition of a data structure in each source file is not maintainable
- What about the function declarations?
 - How a source g.c can call a function implemented in f.c
- Solution: define a **header file**
 - Suffix: .h
 - Contains data structures and function declarations (not the function implementations)
 - A function declaration has to be prefixed by **extern**

Header file

`#include` means
“copy-paste”

```
struct point {  
    int x;  
    int y;  
};  
  
extern struct point* create();
```

point.h

```
#include "point.h"  
  
int main(...) {  
    struct point* p = create();  
    ...  
}
```

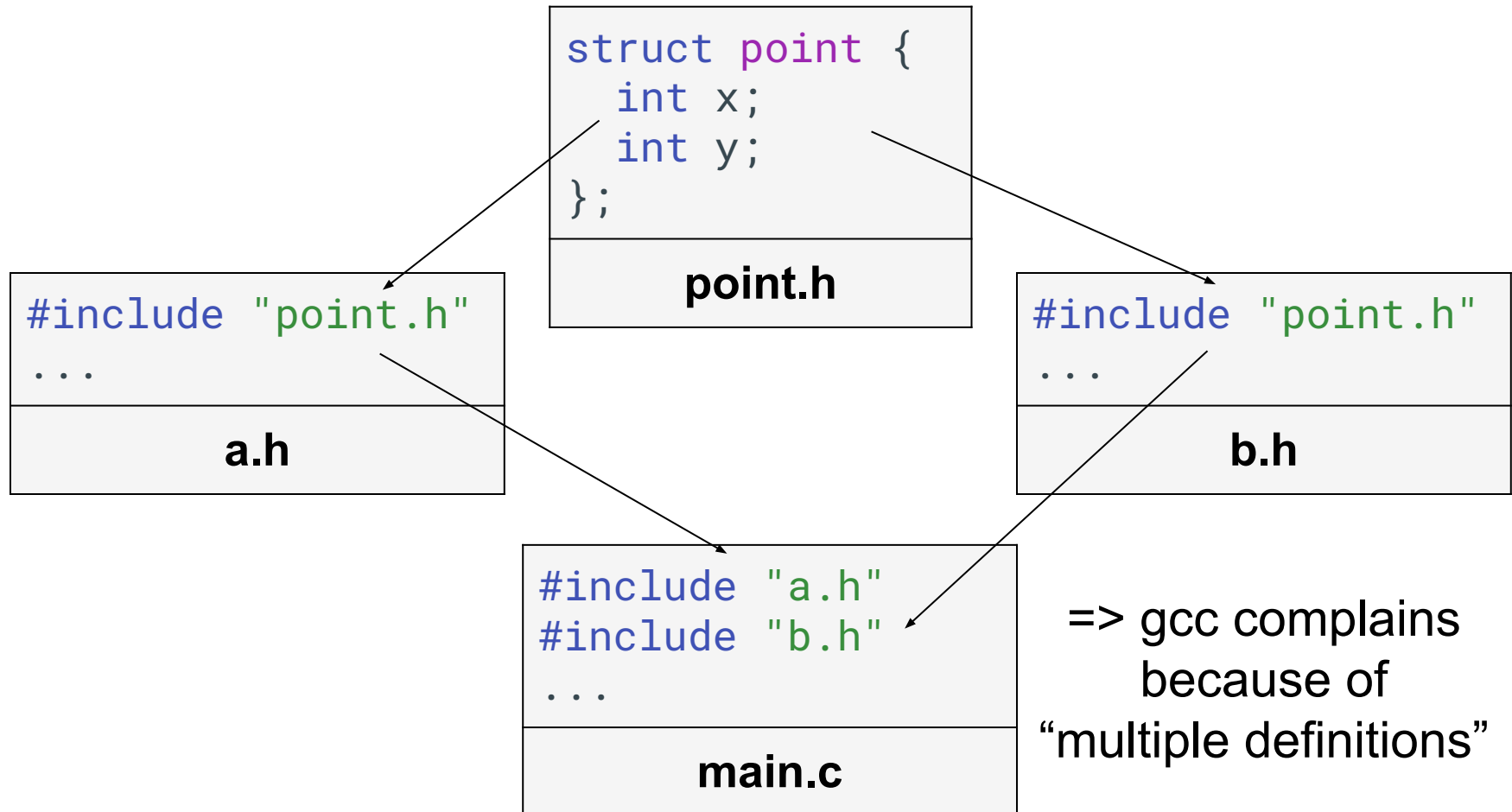
main.c

```
#include "point.h"  
#include <stdlib.h>  
  
struct point* create() {  
    struct point* p =  
        malloc(sizeof(*p));  
    p->x = 0;  
    p->y = 0;  
    return p;  
}
```

point.c

Include once (1/2)

- We can end up with multiple definitions of a structure



Include once (2/2)

- Avoid multiple definitions with `#ifndef/#define/#endif`

```
#ifndef _POINT_H_
#define _POINT_H_

struct point {
    int x;
    int y;
};

#endif
```

point.h

- First include
 - `_POINT_H_` is not defined
 - copy the content
 - define `_POINT_H_`
 - we have the definition
- Second include
 - `_POINT_H_` is already defined
 - ignore the content
 - => a single definition

Static libraries

- Static libraries: used to reuse multiple object files at once
 - Aggregate several object files in an archive
 - `ar rcs libengine.a f1.o f2.o ...`
- Two solutions to link a binary with a static library
 - `gcc -o exe f3.o ../other_project/libengine.a`
 - `gcc -o exe f3.o -lengine -L../other_project/`

Shared library (1/2)

- With a static library, the code is replicated in each process
 - Uselessly consumes memory
- Shared library: share the code between different processes
 - `gcc -shared -o libengine.so f1.c f2.c`
- Link an executable with a shared library
 - `gcc -o exe f3.c -L../other_project -lengine`
- At runtime:
 - The operating system loads the library if it is not loaded yet
 - Share the code from another process if the library is loaded

Shared library (2/2)

- When the operating system loads a binary
 - It searches the shared library in the file system
 - By default in `/lib` and `/usr/lib`
- Sometimes, a shared library is located elsewhere
 - Use the shell variable `LD_LIBRARY_PATH`
- Usage

```
$ LD_LIBRARY_PATH=../other_project  
$ ./my_great_program
```

```
# => search libengine in ../other_project
```

terminal

Key concepts

- Compilation versus linking
 - Compilation: generate an object file from a source file
 - Linking: generate an executable from object files
- Header file (.h)
 - Gather the data structures and function definitions used in multiple source files
 - Guarded with `#ifndef/#define/#endif`
- Libraries: brings a set of object files together
 - Static (.a): included in each executable
 - Shared (.so): shared between multiple processes