Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

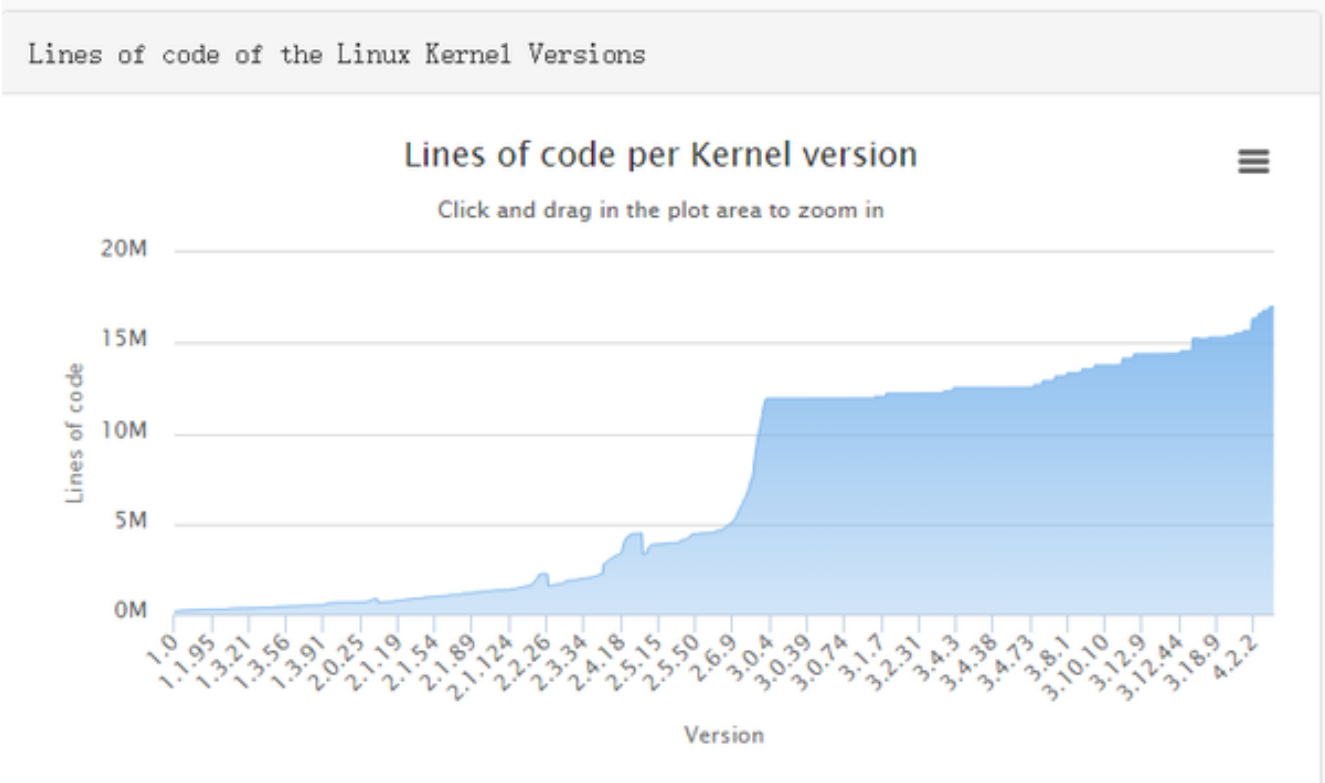
1. Modular programming and libraries in C
   - Modules: overview
   - Modules in C
   - Libraries

Outline

1. Modular programming and libraries in C
   - Modules: overview
   - Modules in C
   - Libraries

Issues with a single long program

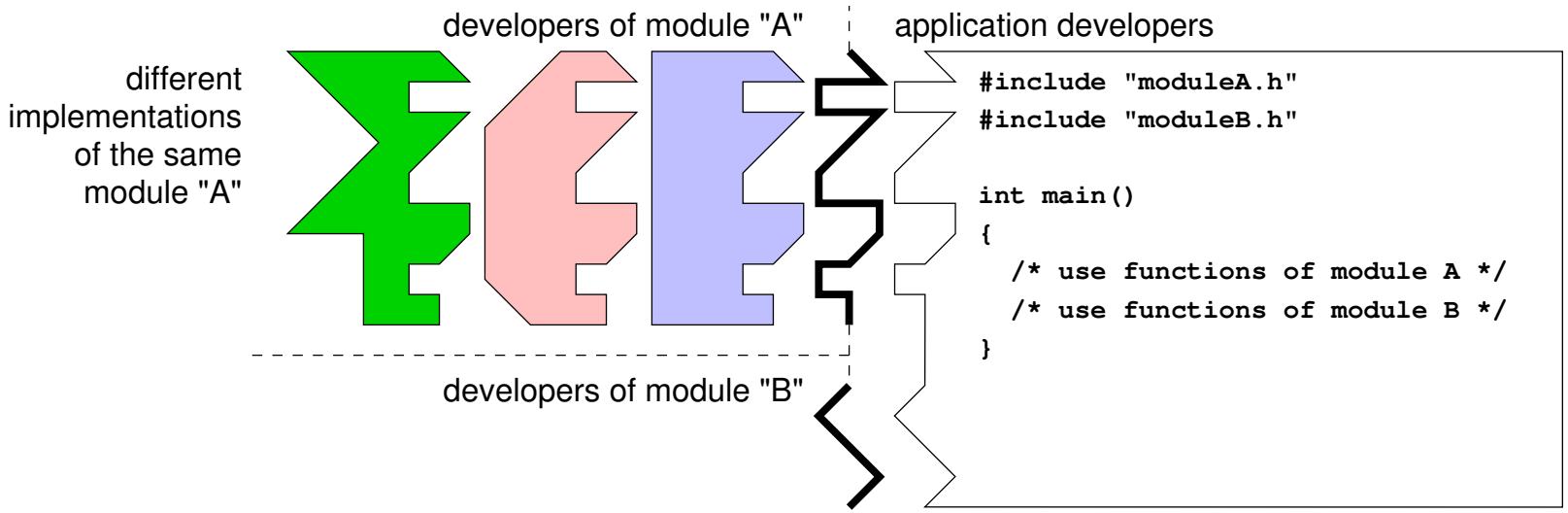
- Large programs (measured in number of lines or number of functions) may require many different functionalities
- Having the entire code on a single file may be problematic



- If a small modification is made on one function the entire file needs to be recompiled

Modules

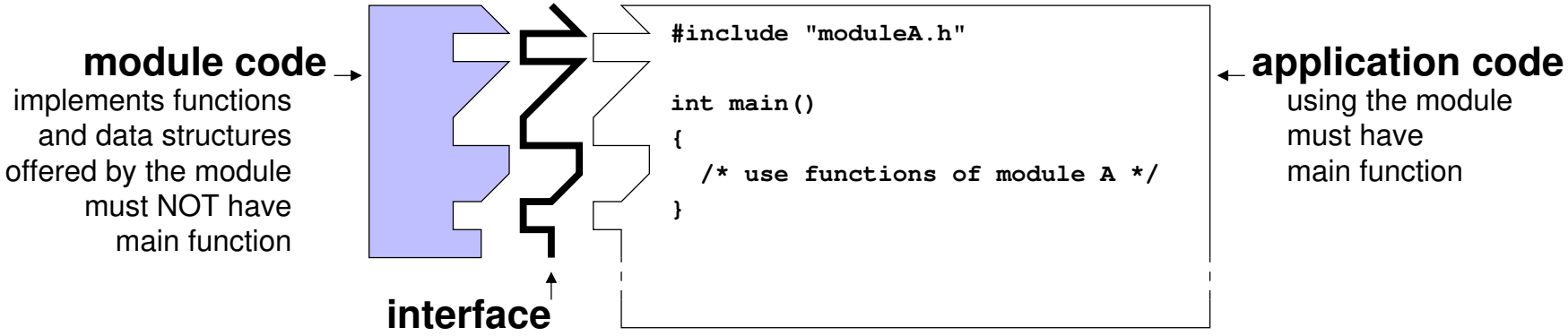
- Solution: to group functions and data that cover a specific functionality in a single source file (a *module*)
  - the "granularity" of a module is similar to the one of *objects* in object-oriented programming
- Development of a large project (example: the Linux kernel):
  1. the large project is split down into smaller "modules" (example: the Linux scheduler, the memory management, the I/O management, ...)
  2. possibly different teams develop each single module
- The *interface* describes the features offered by the module (a sort of "contract" between the developers and the user of the module)



Outline

1. Modular programming and libraries in C
   - Modules: overview
   - Modules in C
   - Libraries

Modules in C



1. **interface**: the *header* file (example: `"moduleA.h"`)
   - lists functions and data types (`typedef`, `struct`,...) of the module
   - it is included by the `#include` directive
   - it is **never** compiled: there is no executable code, no var declaration
2. **module code**: implementation of module (example: `"moduleA.c"`)
   - contains the implementation of the functions listed in the header file
   - may contain global variables read/written by the module code
   - **only compiled** by `gcc -c` to produce the object file
   - does **not** contain a `main()` function
3. **application code** (example: `application.c`)
   - it uses the module by including the header file
   - it **must** contain a `main()` function
   - compiled and **linked** to all used object files

Modules in C: the interface

- The interface of a module in C is the *header file*.
  Example: `module-name.h`
- It is included by all programs using the module by

```
#include "module-name.h"
```

- To avoid multiple inclusion it starts/ends as follows:

```
#ifndef _MODULE_NAME_H
#define _MODULE_NAME_H
/*
 * List of data types and functions offered
 * by the module with EXPLANATORY COMMENTS!!
 */
#endif /* _MODULE_NAME_H */
```

- it doesn't contain any executable code (no assignments, `for`, `if`, ...)
- no variable declaration (except "fake" declarations by `extern`)
- it is **never** compiled, included by others. **Never, ever** write something like:

```
gcc module-name.h
```

Modules in C: the implementation

- It describes how the module is implemented.
  Example: `module-name.c`
- It also includes its own header

```
#include "module-name.h"
/*
 * Implementation of the functions listed in
 * module-name.h
 */
```

- May be less commented: it is read by the module developers, **not** by the module users
- A module is **compiled only** by (notice the flag -c)
  `gcc -c module-name.c`
  which produces the object file `module-name.o` (not an executable)
- The implementation may be **hidden** to the user, who **only** needs
  1. the (text) header file `module-name.h` (to compile his code)
  2. the (binary) object file `module-name.o` (to link his code)

Modules in C: application code

- The application is what you launch from the terminal
- If it wants to use the module, it must include its header file

```
#include "module-name.h"
/* Application dependent functions */

int main() {
  /* Application code */
}
```

- the pre-processor directive `#include "module-name.h"` allows using the module functions and data types and compiling without errors
- the code of the module functions is then added during the linking stage, it is **not** compiled with the application
  - `gcc -c application.c` (`application.c` only compiled)
  - `gcc application.o module-name.o -o application` (linking)
- above steps in one shot
  - `gcc application.c module-name.o -o application`

Example: the "matrix" module

- Let us have a look to a module implementing some matrix operations
  - *matrix.h*, the header file of the module (the interface)
  - *matrix.c*, the implementation of the module
  - *application.c*, an example of code using the module
- The module must then be compiled (**only**) by

```
gcc -c matrix.c
```

- Any program (such as *application.c*) which wants to use the module, must include only

```
#include "matrix.h"
```

and be compiled and linked by

1. `gcc -c application.c` (compiling `application.c`)
2. `gcc application.o matrix.o` (linking all objects)

or

1. `gcc application.c matrix.o`
   (compiles `application.c` and links it to `matrix.o`)

Outline

1. Modular programming and libraries in C
   - Modules: overview
   - Modules in C
   - Libraries

Standard C Library

- When the modules to be used are many, it may become complicated even to write the command line to compile

```
gcc app.c mod1.o mod2.o mod3.o ....
```

- The term *library* is often used to denote a collection of modules
- The Standard C Library (`libc`) is a vast collection of widely used functions (`printf(...)`, etc.)
  - GNU `libc` (`glibc`) is the `libc` developed by the Free Software Foundation (Richard Stallman)
- Libraries are normally stored in `/usr/lib` and sub-dirs
  - `libc` is at `ls -l /usr/lib/x86_64-linux-gnu/libc.a`
- By linking with `-l<name>` (such as `-lm`), the content of the library `lib<name>.a` is searched for objects to be linked
  - `libc` is linked always even if not explicitly specified

Libraries: statis vs. dynamic

- A library may be:
  - **static**, when the linked code is embedded into the executable (larger executable files, but standalone)
  - **dynamic**, when the executable does not contain the linked code. The code of linked functions is in different executable segments
- Static libraries
  - are loaded at **linking time**
  - have names `lib<name>.a`
- Dynamic libraries:
  - are loaded at **run time**
  - have names `lib<name>.so` (so=shared object)
- Normally, libraries are provided in both forms

```
ls -l /usr/lib/x86_64-linux-gnu/libc.*
```

- The linker `gcc`
  - uses dynamic libraries if available
  - uses static libraries if dynamic libraries unavailable or `-static` specified at linking time
- `gcc test-fun-prt.c` vs. `gcc test-fun-prt.c -static`

Static libraries: the ar utility

1. The `ar` utility is used to **ar**chive many single files in a unique one
2. In programming, `ar` is used to store many object files into a unique *library*
3. Example: show the content of the Standard C Library (`libc`) by

```
ar t /usr/lib/x86_64-linux-gnu/libc.a | less
```

4. Example: extract one object file by

```
ar x /usr/lib/x86_64-linux-gnu/libc.a printf.o
```

Object dump

- `objdump` shows the content of an object file
- The format used to show the object is an ELF (Executable and Linkable Format) file
- Examples
  1. to see the assembly code of the module `matrix`, try
     ```
     objdump -d matrix.o
     ```
  2. recompilie by
     ```
     gcc -c -g matrix.c
     ```
     and then try the next command to see source code and assembly
     ```
     objdump -S matrix.o
     ```
- `objdump` may be used for reverse engineering on executables: understanding from the binaries what the program is doing
- Example: show the assembly code of the printf by
  ```
  objdump -d printf.o
  ```