Environnements et Outils de Développement
Cours 4 — Linking

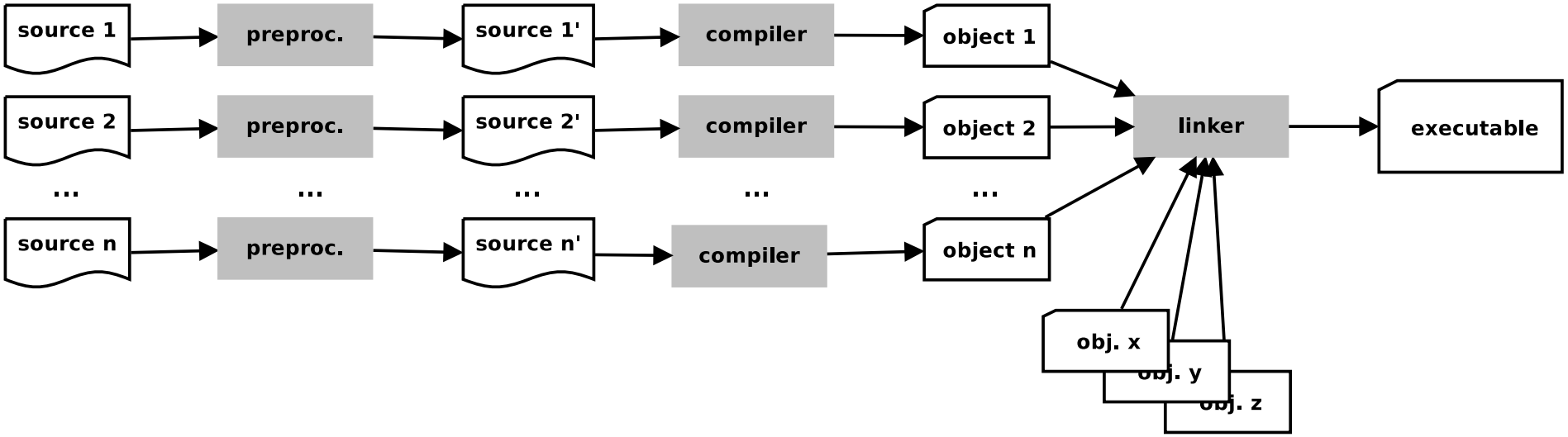
Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot - Paris 7

URL http://upsilon.cc/~zack/teaching/1112/ed6/

Sommaire

The build process (reminder)



Sommaire

Anatomy of a C source file

A C source file usually consists of one or more of :

definition that associates a name to a body given in the file

declaration that states that a name exists somewhere else in the (final) program

There are two kinds of names :

- function
- variable

	Function	**Variable**
Declaration	`int foo(int x);`	`extern int x;`
Definition	`int bar(int x) { ... }`	`int x = 42;`

Anatomy of a C source file (cont.)

- variable definition

```
int x = 42;
```

I hereby introduce a variable named x of type `int`

  - optional : with this visibility (e.g. `static`)
  - optional : with this initial value (e.g. 42)

- variable declaration

```
extern int x;
```

I use in this file a variable called x of type `int`. I promise it can be found elsewhere

Anatomy of a C source file (cont.)

- function definition
  `int bar(int x) {...}`

  I hereby introduce a function called `bar` that takes an `int` and return an `int`, with body `{...}`

  - optional : with this visibility (e.g. `static`)

- function declaration
  `int foo(int x);`

  I use in this file a function called `foo` that takes an `int` and return an `int`. I promise it can be found elsewhere

Anatomy of a C source file — example

```c
/* Definition of an initialized global variable */
int x_global = 1;

/* Declaration of a global variable that exists somewhere else */
extern int y_global;

/* Declaration of a function that exists somewhere else */
int fn_a(int x, int y);

/* Definition of a function. */
int fn_b(int x) { return (x+1); }

/* Definition of another function. */
int fn_c(int x_local) {
        /* Definition of an initialized local variable */
        int y_local = 3;

        /* Code that refers to local and global variables and other
         * functions by name */
        x_global = fn_a(x_local, x_global);
        y_local = fn_a(x_local, y_global);
        return (x_global + y_local);
}
/* end of anatomy.c */
```

Symbols through compilation

When we compile C sources to objects, names will be transformed to symbols, depending on their kind :

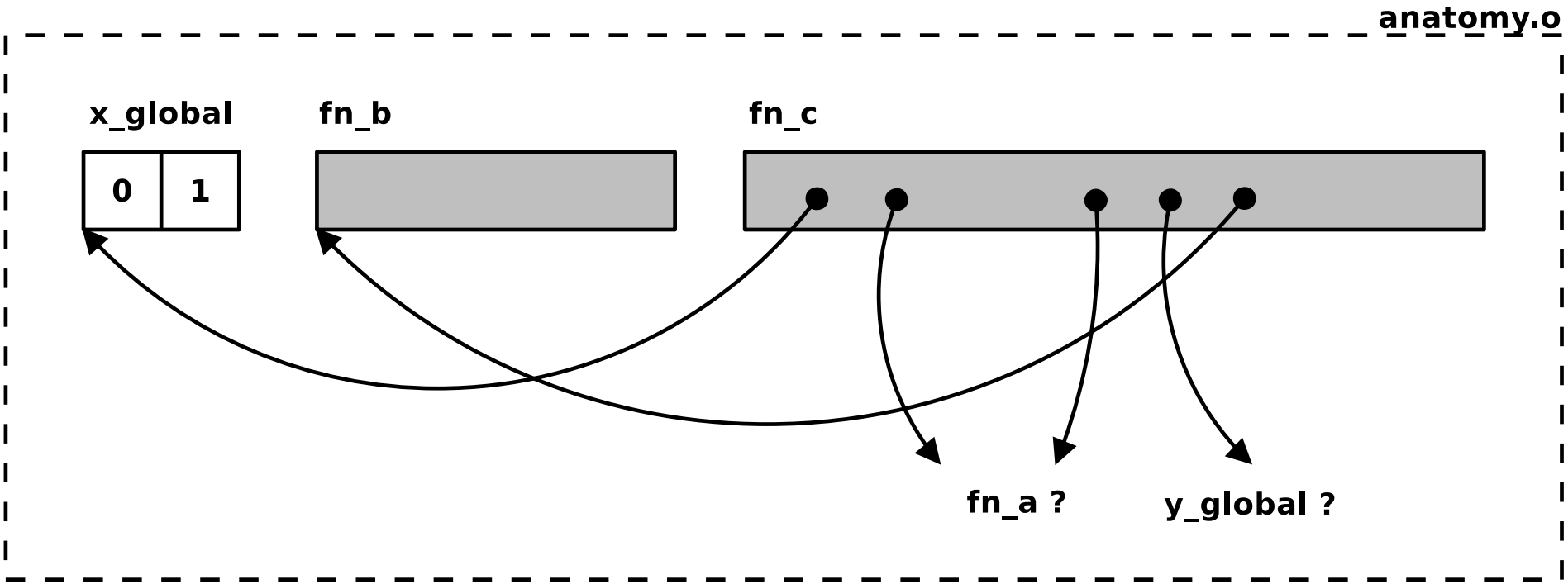
Definitions

- variable definitions → defined variables
  - space to hold the variable (possibly set to the initial value)
- function definitions → defined functions
  - machine code that implements the function body when executed

Declarations

- variable declarations → undefined symbols
- function declarations → undefined symbols
  - they remember our promises that homonymous variables/functions exist somewhere else in the program
  - all references in the object to undefined symbols will remain dangling

Symbols through compilation (cont.)



Looking at symbols with nm

Using the nm utility we can inspect the symbols of object files.

nm output is tabular and uses single letters to describe the status of symbols. Some of them are :

- 'U' undefined symbol (pointing to variable or function)
- 'T'/'t' machine code of a function defined in the object (historical mnemonic for "**t**ext")
- 'D/'d' initialized variable in the object (mnemonic for "**d**ata")
- 'B/'b' uninitialized variable in the object (historical mnemonic for "**b**ss")

Case changes according to visibility : lowercase is for local symbols, uppercase for global ones.

nm — example

```c
/* Definition of an initialized global variable */
int x_global = 1;

/* Declaration of a global variable that exists somewhere else */
extern int y_global;

/* Declaration of a function that exists somewhere else */
int fn_a(int x, int y);

/* Definition of a function. */
int fn_b(int x) { return (x+1); }

/* Definition of another function. */
int fn_c(int x_local) {
        /* Definition of an initialized local variable */
        int y_local = 3;

        /* Code that refers to local and global variables and other
         * functions by name */
        x_global = fn_a(x_local, x_global);
        y_local = fn_a(x_local, y_global);
        return (x_global + y_local);
}
/* end of anatomy.c */
```

nm — example (cont.)

```
$ gcc -Wall -c anatomy.c
$ nm anatomy.o
                 U fn_a
0000000000000000 T fn_b
000000000000000f T fn_c
0000000000000000 D x_global
                 U y_global
```

nm — example (cont.)

```
$ gcc -Wall -c anatomy.c
$ nm anatomy.o
                 U fn_a
0000000000000000 T fn_b
000000000000000f T fn_c
0000000000000000 D x_global
                 U y_global
```

What would happen with `gcc -Wall anatomy.c`?

nm — example (cont.)

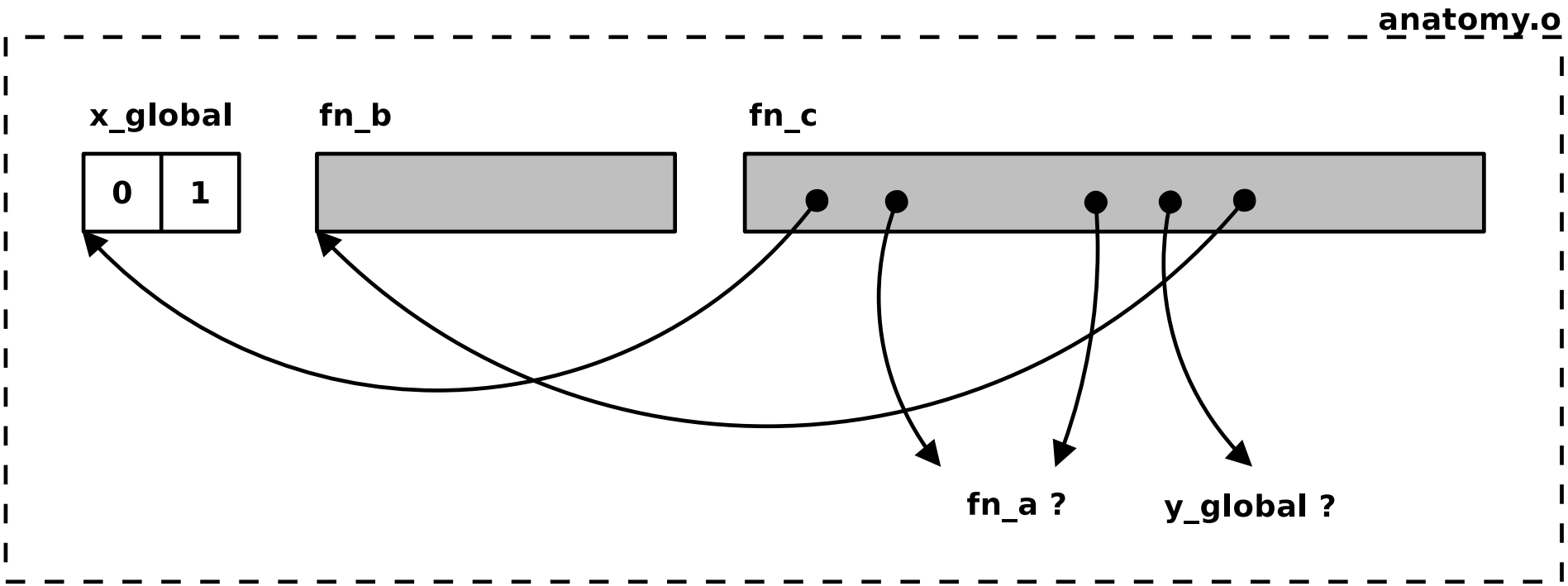
```
$ gcc -Wall -c anatomy.c
$ nm anatomy.o
                 U fn_a
0000000000000000 T fn_b
000000000000000f T fn_c
0000000000000000 D x_global
                 U y_global
```

```
$ gcc -Wall anatomy.c
/usr/lib/gcc/x86_64-linux-gnu/4.6/../../../x86_64-linux-gnu/crt1.o:
In function ‘_start’: (.text+0x20): undefined reference to ‘main’
/tmp/ccoZOIyL.o: In function ‘fn_b’:
anatomy.c:(.text+0x2f): undefined reference to ‘fn_a’
anatomy.c:(.text+0x3b): undefined reference to ‘y_global’
anatomy.c:(.text+0x47): undefined reference to ‘fn_a’
collect2: ld returned 1 exit status
$
```

Sommaire

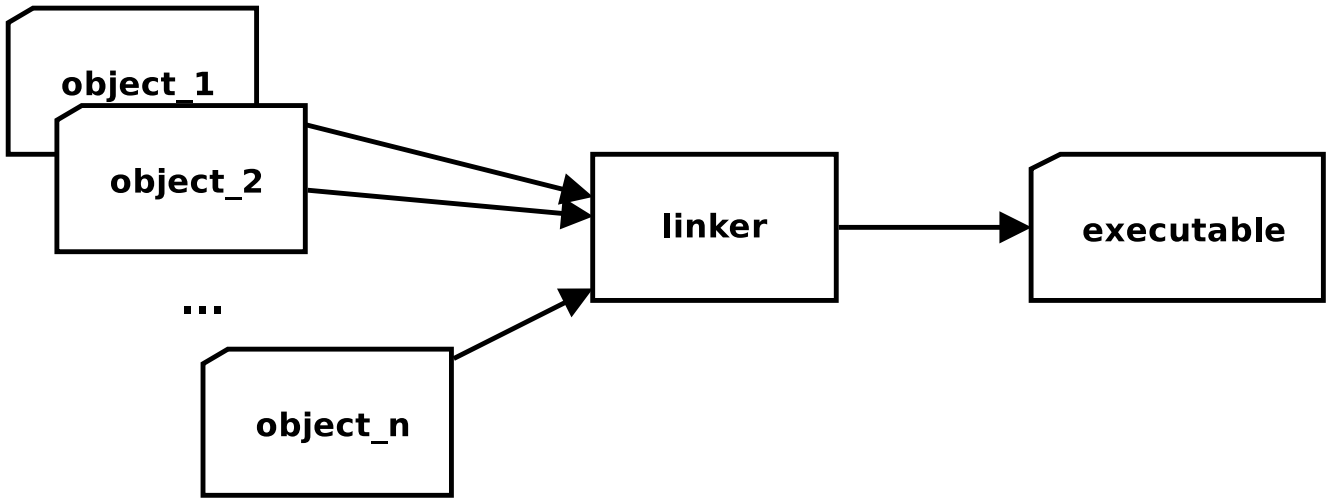
Where the C compiler stops

After compilation an object file looks like this :



- we will not be able to obtain a running program until all undefined symbols have been fixed

What the linker does



1. ensure that promises about symbols that "can be found elsewhere" are respected
   - i.e. within the linked objects, all undefined symbols shall correspond to existing definitions
2. link (hence the name) dangling references to the corresponding definitions
3. assemble all objects together in an executable
   - whose execution will start from the `main` symbol

The linker algorithm — intuition

Reminder

```
$ gcc -o myprogram object_1.o ...object_n.o
```

1. collect defined symbols of all objects in a set D
2. collect undefined symbols of all objects in a set U
3. $U \leftarrow U \cup \{\texttt{main}\}$
4. if $U \nsubseteq D$ i.e. some definitions are missing
   - FAIL
5. if $U \subseteq D$ i.e. all promises have been met
   - link each undefined symbol with its definition
   - generate executable

Linking — example (anatomy.c)

Defined symbols :

- fn_b
- fn_c
- x_global

Undefined symbols :

- fn_a
- y_global

Linking — example (utils.c)

```c
/* global variable definition used from elsewhere */
int y_global = 42;

/* global function definition used from elsewhere */
int fn_a(int x, int y) {
        return x+y;
}

/* end of utils.c */
```

Defined symbols :

- fn_a
- y_global

Undefined symbols : none

Linking — example (main.c)

```c
#include <stdio.h>
#include <stdlib.h>

int fn_c(int x);

int main(void) {
        printf("%d\n", fn_c(17));
        exit(EXIT_SUCCESS);
}

/* end of main.c */
```

Defined symbols :

- main

Undefined symbols :

- exit
- fn_c
- printf

Linking — example

```
$ gcc -Wall -c anatomy.c
$ gcc -Wall -c utils.c
$ gcc -Wall -c main.c
$ gcc -o anatomy main.o anatomy.o utils.o
$ ./anatomy
77
$
```

Linking — example

```
$ gcc -Wall -c anatomy.c
$ gcc -Wall -c utils.c
$ gcc -Wall -c main.c
$ gcc -o anatomy main.o anatomy.o utils.o
$ ./anatomy
77
$
```

But where do `printf` and `exit` come from?

Libraries

- many programs will need to do the same sort of things
  - e.g. printing, allocating memory, parsing XML files, etc.
- we do not want to implement those features again and again
- we want to store them "somewhere" and link with that somewhere all code that needs it

Library

A library is a set of objects assembled together and stored in an accessible place known by the linker.

- on UNIX-like systems libraries are usually named `libsomething`
- we can ask the linker to link with the whole content of a library passing the `-lsomething` flag on the command line

The C standard library

One special library is linked in by default : the C standard library

- it implements frequently needed features such as printing, memory allocation, OS interaction, etc.
- the library is called `libc` (for "C library")

Example

In our example `printf` and `exit` are undefined symbols among our objects ; they are defined in `libc`.
Given `libc` is linked by default, the following :

```
$ gcc -o anatomy main.o anatomy.o utils.o
```

is equivalent to the more explicit :

```
$ gcc -o anatomy main.o anatomy.o utils.o -lc
```

Linking order

- in fact, the linker processes objects from left to right
- undefined symbols in some object must correspond to definition found in objects to their right

A typical linker invocation should look like :

```
$ gcc main.o high-level.o low-level.o -llib1 -llib2
```

different linking orders (e.g. putting a library before an object that uses it) might result in undefined reference errors !

Sommaire

Maintenance

Consider this :

- you have a library to parse emails
- that is linked in several applications
  - a mail application,
  - a webmail,
  - a mail indexer,
  - an automatic responder,
  - etc.
- one day, a serious security issue is discovered in the email parsing code that allows to execute code on the machine by crafting a particular email
- a few hours later, a fix for the library code is found

How difficult it is to fix all applications that use the library?

Maintenance (cont.)

How difficult it is to fix all applications that use the library?

There are essentially two possibilities :

1. either you have to re-link (and re-install) all applications to have them fixed
2. or it is enough to re-build (and re-install) the library only to have all applications fixed

Static vs dynamic linking

Which of the options will be in effect depends on whether the library has been statically or dynamically linked.

static linking when an object is statically linked, its symbols are copied in the final executable

dynamic linking when an object is dynamically linked, the final executable will maintain a (dangling) reference to that object that will need to be resolved before being able to run the executable

- i.e. the final step of linking is no longer performed at link-time, it is delayed until run-time

Static vs dynamic linking — discussion

Static linking

- advantage : we do not need to separately install the objects that are statically linked, because they *are part* of the program
- disadvantage : we copy duplicate code into different applications
  - executables are larger
  - to fix code in the statically linked objects we need to re-do the linking

Dynamic linking

- advantage : we can change dynamically linked code only once, for all programs that use it
- disadvantage : we need to install on the target machine both the executables and the dynamically linked objects

ldd — example

Using the ldd utility we can inspect which objects will need to be dynamically linked at runtime for a given program.

```
$ gcc -o anatomy main.o anatomy.o utils.o
$ ldd anatomy
        linux-vdso.so.1 =>  (0x00007fff82fa2000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fad95bc4
        /lib64/ld-linux-x86-64.so.2 (0x00007fad95f74000)
```

By default, the C standard library is dynamically linked. Indeed, the symbols coming from it remain undefined even in the final executable :

```
$ nm anatomy
[...]
0000000000400534 T main
                 U printf@@GLIBC_2.2.5
0000000000600a30 D x_global
0000000000600a34 D y_global
$
```

Static linking

Traditionally on UNIX, libraries passed to the linker with `-l` are linked dynamically to maximize code sharing on a given machine.

We can request static linking passing the `-static` flag to the linker : all libraries coming *after* it on the command like will be linked statically rather than dynamically.
`-static` will also request to statically link the C standard library (even if it does not appear on the linker command line by default).

Static linking — example

```
$ gcc -o anatomy main.o anatomy.o utils.o -static
$ ls -l anatomy
$ -rwxr-xr-x 1 zack zack 776371 feb 29 11:43 anatomy
$
```

The executable is much bigger now(!), because all of the C standard library has been *copied* into it. Also :

- it no longer references dynamically linked objects,
- nor has undefined symbols

```
$ ldd anatomy
        not a dynamic executable
$ nm anatomy
[...]
0000000004010a0 T printf
[...]
```

Sommaire

Shared libraries

Libraries that are meant to be dynamically linked are called shared libraries.

- the details of how to build (and maintain) a shared library is a complex topic, which is outside of the scope of this class
- we just give an overview of the main steps of how to build a shared library and load use it in your programs

Shared library — example (library implementation)

We build a hello library, meant to "make it easier" to greet your users. It is composed by two files (interface and implementation) :

- hello.h :

```
#ifndef __HELLO_H__
#define __HELLO_H__

void hello(void);

#endif /* __HELLO_H__ */
```

- hello.c :

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

void hello(void) {
        printf("Hello, world!\n");
}
```

Shared library — example (using the library)

We can use the library as usual, without having to care whether it will be statically or dynamically linked.

Here is how we will use the library from the sample program hello-test.c :

```
#include <stdlib.h>
#include "hello.h"

int main(void) {
        hello();
        exit(EXIT_SUCCESS);
}
```

Shared library — example (building the library)

To build the library we must take care of two things :

- tell the compiler the objects are meant to become part of a shared library passing `-fPIC`
- tell the linker we are creating a shared library passing `-shared`

```
$ gcc -Wall -fPIC -c hello.c
$ gcc -shared -o libhello.so hello.o
$ ls -al libhello.so
-rwxr-xr-x 1 zack zack 6037 feb 29 12:02 libhello.so
$
```

Shared library — example (linking with the library)

Linking with the library is done as usual, but we need to tell the linker where to find `libhello` (passing `-L`*`directory`*), because it is not installed in the default library location.

```
$ gcc -L/home/zack/ed6/git/tp-2/libhello -o hello-test hello-test.c
$ ldd hello-test
        linux-vdso.so.1 =>  (0x00007fff533c5000)
        libhello.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f83b386
        /lib64/ld-linux-x86-64.so.2 (0x00007f83b3c19000)
```

Shared library — example (running the program)

Once more, given the library is not installed in the default library location, we need to tell the dynamic linker where to find the library before running the program. We can do so setting the LD_LIBRARY_PATH environment variable.

```
$ ./hello-test
./hello-test: error while loading shared libraries:
libhello.so: cannot open shared object file: No such file or directo

$ export LD_LIBRARY_PATH=/home/zack/ed6/git/tp-2/libhello
$ ./hello-test
Hello, world!
$
```

References

- David Drysdale, *Beginner's Guide to Linkers*
  `http://www.lurklurk.org/linkers/linkers.html`
  (examples from these slides adapted from it, under GFDL)
- John Levine, *Linkers and Loader*
  Morgan Kaufmann ; 1 edition, 1999