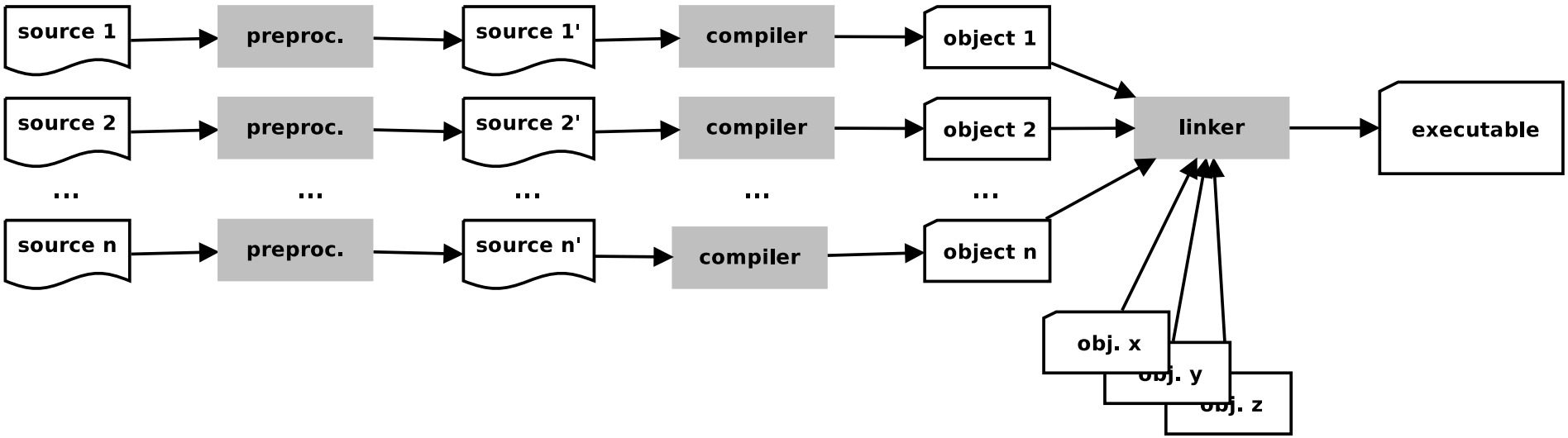
Conduite de Projet
Cours 6 — Linking

Stefano Zacchiroli
zack@irif.fr

Laboratoire IRIF, Université Paris Diderot

2018–2019

The build process (reminder)



Outline

Outline

Anatomy of a C source file

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

definition an association between a *name* and a *body*; the body is provided in the file itself

declaration a statement that a name is defined somewhere else in the (final) program

There are two kinds of names:

- function names
- variable names

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

Anatomy of a C source file (cont.)

- variable definition

  `int x;`

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

  `int x = 42;`

  - 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 `{...}`
  - note: the argument name (x) isn't part of the signature

- 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_b(y_global);
        return (x_global + y_local);
}
/* end of anatomy.c */
```

Symbols through compilation

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

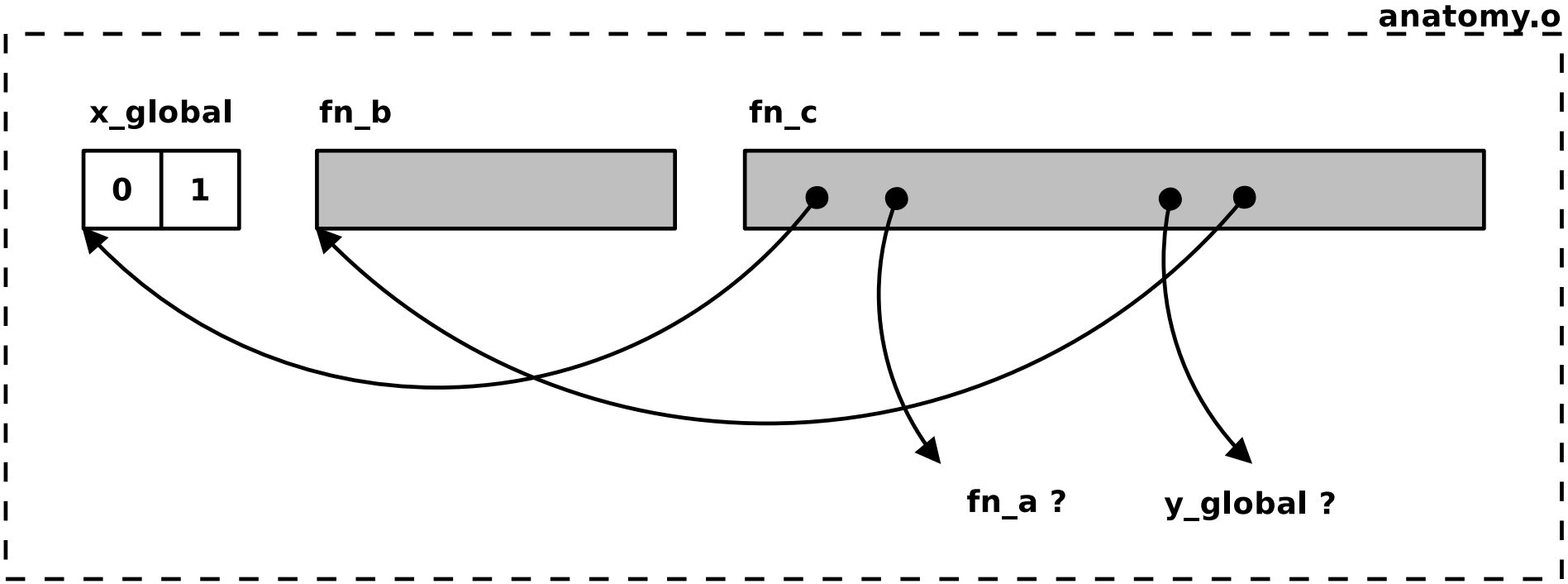
Definitions

- 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 represent our promises that matching defined symbols exist somewhere else in the program
  - all references in the object to undefined symbols will remain dangling

Symbols through compilation (cont.)




```
int x_global = 1;
extern int y_global;
int fn_a(int x, int y);
int fn_b(int x) { return (x+1); }
int fn_c(int x_local) {
        int y_local = 3;
        x_global = fn_a(x_local, x_global);
        y_local = fn_b(y_global);
        return (x_global + y_local);
}
```

Looking at symbols with nm

Using the nm tool 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 a variable or function)
- ‘T’/‘t’ machine code of a function defined in the object (historical mnemonic for “**t**ext”) [1]
- ‘D/‘d’ initialized variable in the object (mnemonic for “**d**ata”)
- ‘B/‘b’ uninitialized variable in the object (historical mnemonic for “**b**ss”)

See: `man nm`

1. upper/lowercase denotes symbol visibility, more on this later

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
```

```
int x_global = 1;
extern int y_global;
int fn_a(int x, int y);
int fn_b(int x) { return (x+1); }
int fn_c(int x_local) {
        int y_local = 3;
        x_global = fn_a(x_local, x_global);
        y_local = fn_b(y_global);
        return (x_global + y_local);
}
```

nm — example (cont.)

```
int x_global = 1;
extern int y_global;
int fn_a(int x, int y);
int fn_b(int x) { return (x+1); }
int fn_c(int x_local) {
        int y_local = 3;
        x_global = fn_a(x_local, x_global);
        y_local = fn_b(y_global);
        return (x_global + y_local);
}
```

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

nm — example (cont.)

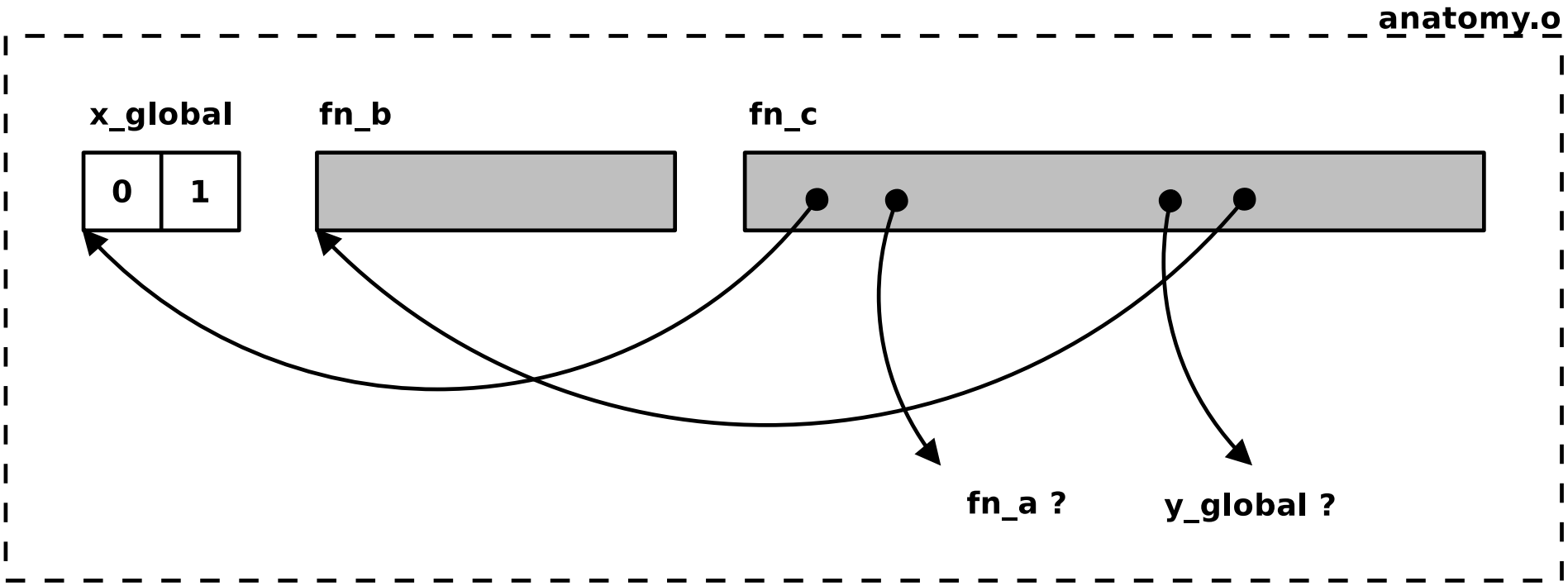
```
int x_global = 1;
extern int y_global;
int fn_a(int x, int y);
int fn_b(int x) { return (x+1); }
int fn_c(int x_local) {
        int y_local = 3;
        x_global = fn_a(x_local, x_global);
        y_local = fn_b(y_global);
        return (x_global + y_local);
}
```

```
$ 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_c’:
anatomy.c:(.text+0x2f): undefined reference to ‘fn_a’
anatomy.c:(.text+0x3b): undefined reference to ‘y_global’
collect2: ld returned 1 exit status
$
```

Outline

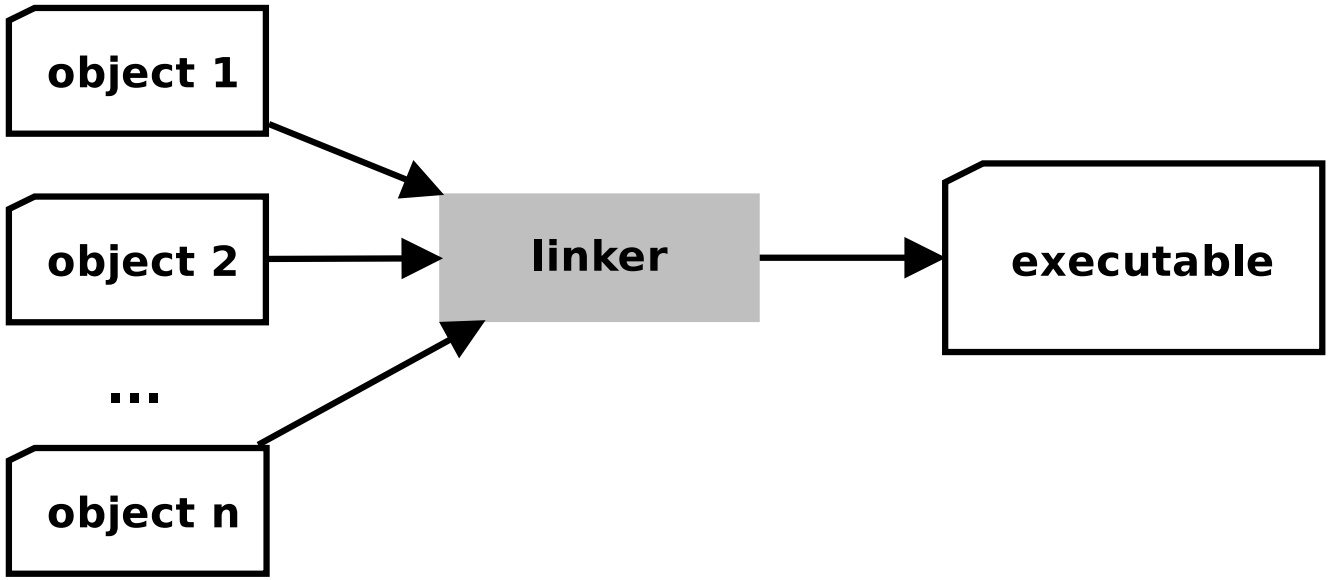
Where the C compiler stops

After compilation an object file looks like this:



we will not be able to obtain a running program until all undefined symbols have been “resolved”

What the linker does



1. ensure that promises about symbols that "can be found elsewhere" are respected
   - i.e., among *all* 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 as an algorithm (sketch)

Reminder

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

0. let O be the set of all object files being linked
1. $\forall o_i \in O$ collect its defined symbols in a set D_i
   - if $D_i \cap D_j \neq \emptyset$ for some $i \neq j$ *multiple symbol definitions*
   - return FAIL
2. $D \leftarrow \bigcup D_i$ *the set of all defined symbols*
3. collect all undefined symbols from all objects in a set U
4. $U \leftarrow U \cup \{\texttt{main}\}$
5. if $U \not\subseteq D$ *some definitions are missing*
   - return FAIL

   else (i.e., $U \subseteq D$) *all promises are met*
   - link each undefined symbol with its definition
   - generate executable

Linking — example (anatomy.c)

```
int x_global = 1;
extern int y_global;
int fn_a(int x, int y);
int fn_b(int x) { return (x+1); }
int fn_c(int x_local) {
        int y_local = 3;
        x_global = fn_a(x_local, x_global);
        y_local = fn_b(y_global);
        return (x_global + y_local);
}
```

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

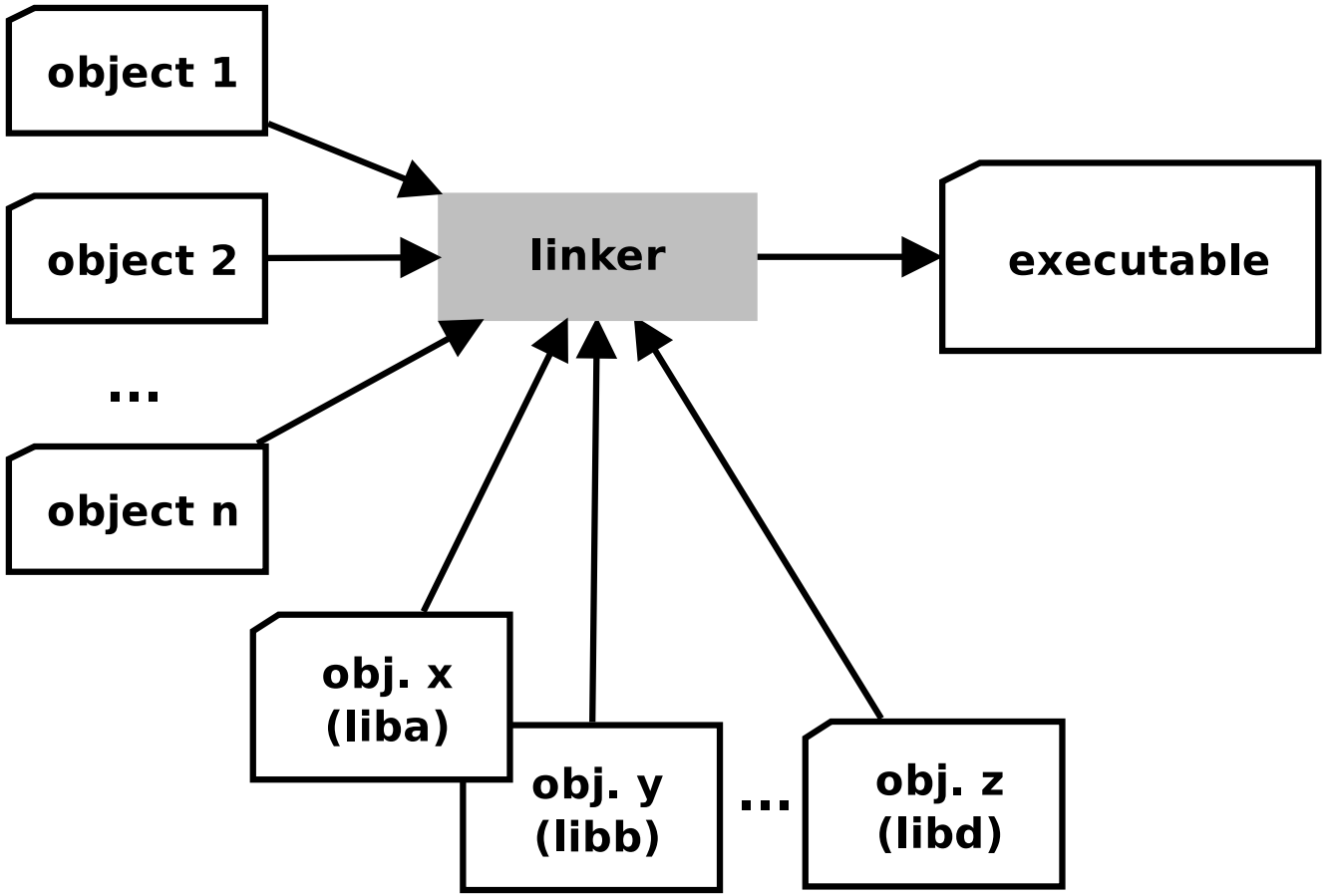
```
$ gcc -Wall -c anatomy.c
$ gcc -Wall -c utils.c
$ gcc -Wall -c main.c
$ gcc -o anatomy main.o anatomy.o utils.o
$ ./anatomy
77
$
```

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?

What the linker does (with libraries)



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 them

Definition (library)

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

- on UNIX-like system libraries are usually named `lib`*`something`*
- we can ask the linker to link with the (entire content of) a library passing the `-l`*`something`* 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

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

A typical linker invocation should look like:

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

i.e., a topological visit on the architecture graph.

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

Outline

Information hiding

Definition (information hiding)

The principle of segregating ("hiding") the design decisions of a software component that are most likely to change, so that *other* software components cannot depend on them.

Intuition: software components offer "services" to other ("client") software components; the "terms of service" are codified in their public APIs.

Goal: minimize the size of the public API of each component.
Because:

- dependency → might require adaptation upon change
- no dependency → no need to adapt upon change

An Abstract Data Type in C — evenint.h

```c
// int guaranteed to be even, i.e., val % 2 == 0
typedef struct even_int even_int;

// create an even int, initialized to 0
even_int *even_int_new(void);

// destroy an even int, freeing memory
void even_int_del(even_int *i);

// getter
int even_int_get(even_int *i);

// setter; will fail with assertion if val % 2 != 0
void even_int_set(even_int *i, int val);

// increase an even int by 2 (in place)
void even_int_incr(even_int *i);

// multiply an even int by a factor (in place)
void even_int_mult(even_int *i, int factor);
```

An Abstract Data Type in C — evenint.c I

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

#include "evenint.h"

struct even_int {
        int val;  // invariant: val % 2 == 0
};

even_int *even_int_new(void) {
        even_int *i = NULL;

        i = malloc(sizeof(struct even_int));
        i->val = 0;

        return i;
}
```

An Abstract Data Type in C — evenint.c II

```
void even_int_del(even_int *i) {
        free(i);
}

void _even_int_set(even_int *i, int val) {
        i->val = val;
}

void even_int_set(even_int *i, int val) {
        assert(val % 2 == 0);
        _even_int_set(i, val);
}

int even_int_get(even_int *i) {
        return i->val;
}
```

An Abstract Data Type in C — evenint.c III

```
void even_int_incr(even_int *i) {
        even_int_set(i, even_int_get(i) + 2);
}

void even_int_mult(even_int *i, int factor) {
        even_int_set(i, even_int_get(i) * factor);
}

// end of evenint.c
```

An Abstract Data Type in C — test

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

void even_int_print(even_int *i) {
        printf("%d\n", even_int_get(i));
}

int main() {
        even_int *i = even_int_new();
        even_int_print(i);
        even_int_incr(i);
        even_int_print(i);
        even_int_mult(i, 21);
        even_int_print(i);
        // even_int_set(i, 43);  // failed assertion
        _even_int_set(i, 43);    // ouch
        even_int_print(i);
}

// end of evenint-test.c
```

Symbol visibility

```
$ gcc -Wall -c evenint.c
$ gcc -Wall -o evenint-test evenint-test.c evenint.o
evenint-test.c: In function ‘main’:
evenint-test.c:16:2: warning:  implicit declaration of
  function ‘_even_int_set’ [-Wimplicit-function-declaration]

$ nm evenint.o
                 U __assert_fail
0000000000000024 T even_int_del
000000000000009c T even_int_get
00000000000000ac T even_int_incr
00000000000000d8 T even_int_mult
0000000000000000 T even_int_new
000000000000003f T _even_int_set
0000000000000056 T even_int_set
                 U free
                 U malloc
```

Symbol visibility (cont.)

- you: *"let's hide this dangerous, low-level, invariant-breaker setter"*
- compiler: *"I can't find a declaration for* `_even_int_set`*, *shrug* "*
- linker: *"oh, here is the definition for* `_even_int_set`*, let's link it to the caller"*

Symbol visibility (cont.)

- you: *"let's hide this dangerous, low-level, invariant-breaker setter"*
- compiler: *"I can't find a declaration for* `_even_int_set`*, *shrug* "*
- linker: *"oh, here is the definition for* `_even_int_set`*, let's link it to the caller"*

BOOM.

The `static` keyword

The `static` keyword in C has several implications, two of which are relevant here:

static global variable do not export the corresponding defined variable symbol, allowing only internal linkage with it

```
static int x_global = 1;
```

static function (ditto) do not export the corresponding defined function symbol, allowing only internal linkage

```
static int fn_b(int x) {
        return (x+1);
}
```

The `static` keyword

The `static` keyword in C has several implications, two of which are relevant here:

static global variable do not export the corresponding defined variable symbol, allowing only internal linkage with it

```
static int x_global = 1;
```

static function (ditto) do not export the corresponding defined function symbol, allowing only internal linkage

```
static int fn_b(int x) {
        return (x+1);
}
```

let's try again...

Hiding “private methods”

```
...
void even_int_del(even_int *i) {
        free(i);
}

static void _even_int_set(even_int *i, int val) {
        i->val = val;
}

void even_int_set(even_int *i, int val) {
        assert(val % 2 == 0);
        _even_int_set(i, val);
}

...
// end of evenint-hiding.c
```

Symbol (in)visibility

```
$ gcc -Wall -c evenint-hiding.c
$ gcc -Wall evenint-test.c evenint-hiding.o
evenint-test.c: In function ‘main’:
evenint-test.c:16:2: warning: implicit declaration of
  function ‘\_even\_int\_set’ [-Wimplicit-function-declaration]
/tmp/ccsQnASJ.o: In function ‘main’:
evenint-test.c:(.text+0x90): undefined reference to ‘_even_int_set’
collect2: error: ld returned 1 exit status

$ nm evenint-hiding.o
                 U __assert_fail
0000000000000024 T even_int_del
000000000000009c T even_int_get
00000000000000ac T even_int_incr
00000000000000d8 T even_int_mult
0000000000000000 T even_int_new
000000000000003f t _even_int_set
0000000000000056 T even_int_set
                 U free
                 U malloc
```

Symbol (in)visibility (cont.)

In `nm`'s output, case changes according to visibility: lowercase is for internal symbols, uppercase for exported ones.

So, this time:

- you: *"let's hide this dangerous, low-level, invariant breaker setter"*
- compiler: *"I can't find a declaration for `_even_int_set`, *shrug* "*
- linker: *"I can't find a definition for `_even_int_set`"*

Symbol (in)visibility (cont.)

In `nm`'s output, case changes according to visibility: lowercase is for internal symbols, uppercase for exported ones.

So, this time:

- you: *"let's hide this dangerous, low-level, invariant breaker setter"*
- compiler: *"I can't find a declaration for `_even_int_set`, *shrug* "*
- linker: *"I can't find a definition for `_even_int_set`"*
  → **ERROR** (which is what you want)

Outline

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 two is in effect depends on whether the library has been statically or dynamically linked into applications.

static linking when an object is statically linked, its defined symbols are copied into 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, but delayed until run-time instead

Static vs dynamic linking — discussion

Static linking

- pro: we do not need to separately install the objects that are statically linked, because they *are part* of the program
- con: we duplicate code into different applications
  - executables are larger
  - to fix code in the statically linked objects we need to re-do the linking (usually: the build alltogether)

Dynamic linking

- pro: we can change dynamically linked code only once, for all programs that use it
- con: 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 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* the flag on the command like will be linked statically.

`-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
[...]
```

Outline

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 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 with `-fPIC` (for "position independent code")
- tell the linker we are creating a shared library with `-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 currently not installed in the default library location.

```
$ pwd
/home/zack/cproj/git/cours-linking/code/libhello
$ gcc -L$(pwd) -o hello-test hello-test.c -lhello

$ ldd hello-test
linux-vdso.so.1 =>  (0x00007fff533c5000)
libhello.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f83b3869000)
/lib64/ld-linux-x86-64.so.2 (0x00007f83b3c19000)
```

Shared library — example (running the program)

Given that the library is not installed in the default library location, we need to also 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 directory

$ export LD_LIBRARY_PATH=$(pwd)
$ ldd hello-test | grep hello
libhello.so => /home/[...]/libhello/libhello.so (0x00007f37b9b2b000)

$ ./hello-test
Hello, world!
$
```

Outline

ABI compatibility — example (hello2.h)

```
#ifndef __HELLO_H__
#define __HELLO_H__

void hello(char *);

#endif /* __HELLO_H__ */
```

ABI compatibility — example (hello2.c)

```c
#include <stdio.h>
#include "hello2.h"

void hello(char *msg) {
        printf("Hello, %s world!\n", msg);
}
```

ABI compatibility — example (hello-test2.c)

```c
#include <stdlib.h>
#include "hello2.h"

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

ABI compatibility — example

```
gcc -Wall -fPIC -c hello2.c
gcc -Wall -shared -o libhello2.so hello2.o
gcc -Wall -L$(pwd) -o hello-test2 hello-test2.c -lhello2

$ export LD_LIBRARY_PATH=$(pwd)
$ ldd hello-test2 | grep hello
libhello2.so => /home/[\ldots]/libhello/libhello.so (0x00007f78ff33d0

$ ./hello-test2
Hello, foobar world!
$
```

ABI *in*compatibility — example

```
$ cp libhello2.so libhello.so
cp: overwrite 'libhello.so'? y

$ sha1sum libhello*.so
b6fbce8dcc0bb4cc48fb39aafe80ac9b65d453bf  libhello2.so
b6fbce8dcc0bb4cc48fb39aafe80ac9b65d453bf  libhello.so

$ ldd hello-test | grep hello
libhello.so => /home/zack/cproj/git/cours-linking/code/libhello/libh

$ ./hello-test  # old main, new library
```

?

ABI *in*compatibility — example

```
$ cp libhello2.so libhello.so
cp: overwrite 'libhello.so'? y

$ sha1sum libhello*.so
b6fbce8dcc0bb4cc48fb39aafe80ac9b65d453bf  libhello2.so
b6fbce8dcc0bb4cc48fb39aafe80ac9b65d453bf  libhello.so

$ ldd hello-test | grep hello
libhello.so => /home/zack/cproj/git/cours-linking/code/libhello/libh

$ ./hello-test  # old main, new library


[1]    31560 segmentation fault  ./hello-test
```

References

- David Drysdale, *Beginner's Guide to Linkers*
  `http://www.lurklurk.org/linkers/linkers.html`
  (some of the examples in this lecture have been adapted from this guide)
- John Levine, *Linkers and Loader*
  Morgan Kaufmann; 1 edition, 1999