CSCI 210 Systems Programming

Week 7

Static/Dynamic Linking with Libraries

Makefiles

git,GitHub ssh access

Overview

- Linking
  - Static versus Dynamic Linking (slides adapted from Cornell CS4414)
- Makefiles
  - Learning to write your own makefiles
- git

Scenario I

- Libraries can be quite big – some are huge. The memory of your computer can easily be completely filled by copies of libraries – maybe identical ones!

Libtorch_cuda.so is too large (>2GB) - PyTorch Forums

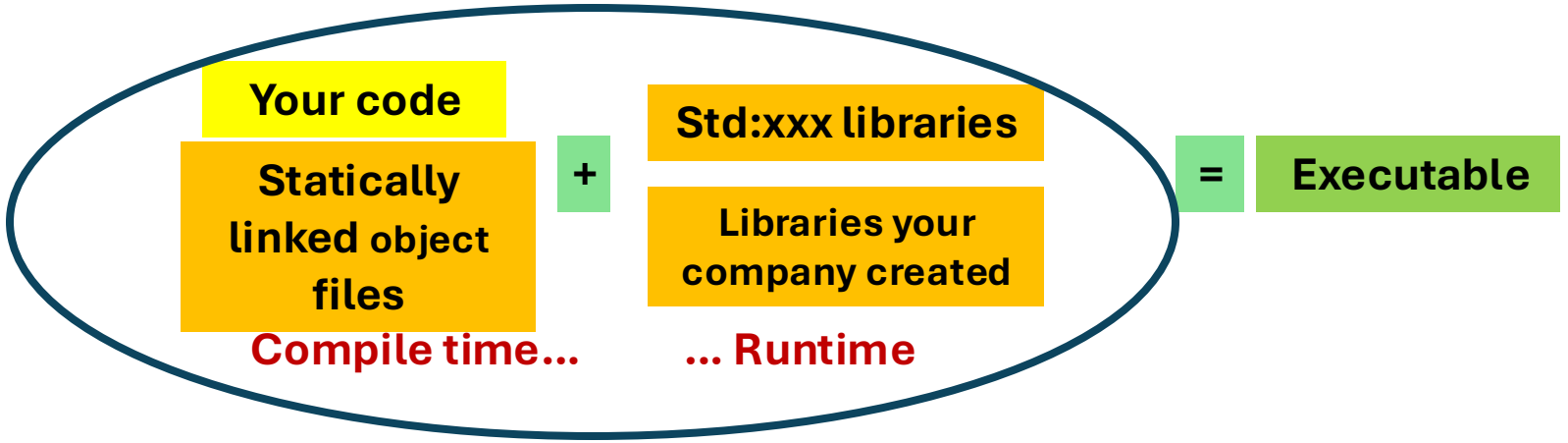
Why is torch wheel so huge (582MB)?

Scenario II

- We are given a system that has pre-implemented programs in it (compiled code plus libraries).

- But now we want to change the behavior of some existing API.
  - Can it be done?

Linking



- A linker takes a collection of object files and combines them into an object file. But this object file will still depend on libraries.
- Next it cross-references this single object file against libraries, resolving any references to methods or constants in those libraries.
- If everything needed has been found, it outputs an executable image.

Example C Program

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

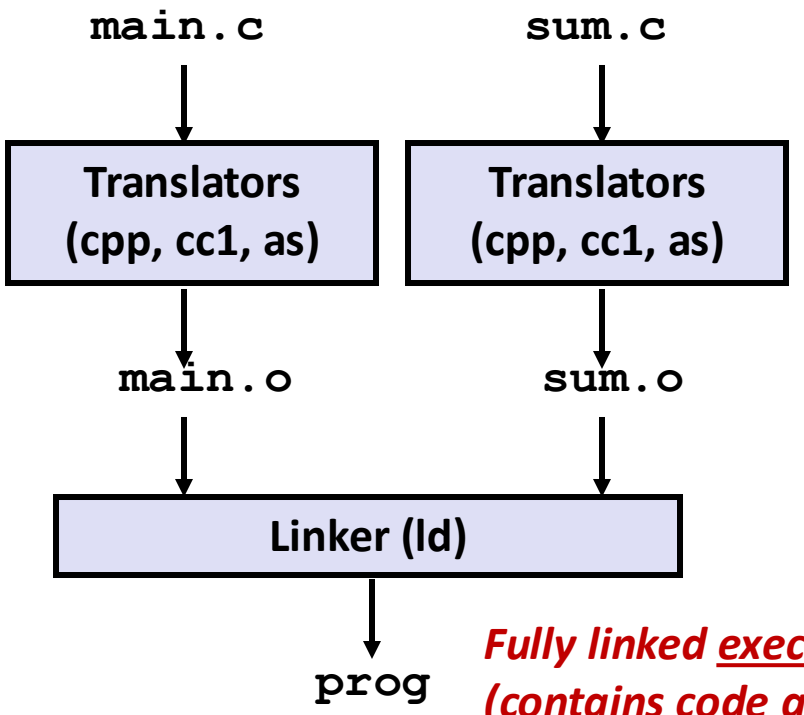
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Linking

- gcc is really a "*compiler driver*": It launches a series of sub-programs
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Why Linkers?

Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass. But later we need to combine all of these.
- Each C source file, .c file, can have implementation of some functions and has its own .h file (declares the type signatures of the functions, define types, etc)

Libraries

- Different programs may need to use similar functionality: e.g., games need a graphics library

An object file is an intermediate form

- An object file contains “incomplete” machine instructions, with locations that may still need to be filled in:
  - Addresses of methods defined in other object files, or libraries
  - Addresses of data and bss segments, in memory

After linking, all the “resolved” addresses will have been inserted at those previously unresolved locations in the object file.

We will skip the details of addressing in this introductory course. You will learn more about these in OS.

How linking works: Symbol resolution

Programs define and reference symbols (global variables and functions):

- void swap() {…} /* define symbol swap */
- swap(); /* reference symbol swap */
- int *xp = &x; /* define symbol xp, reference x */

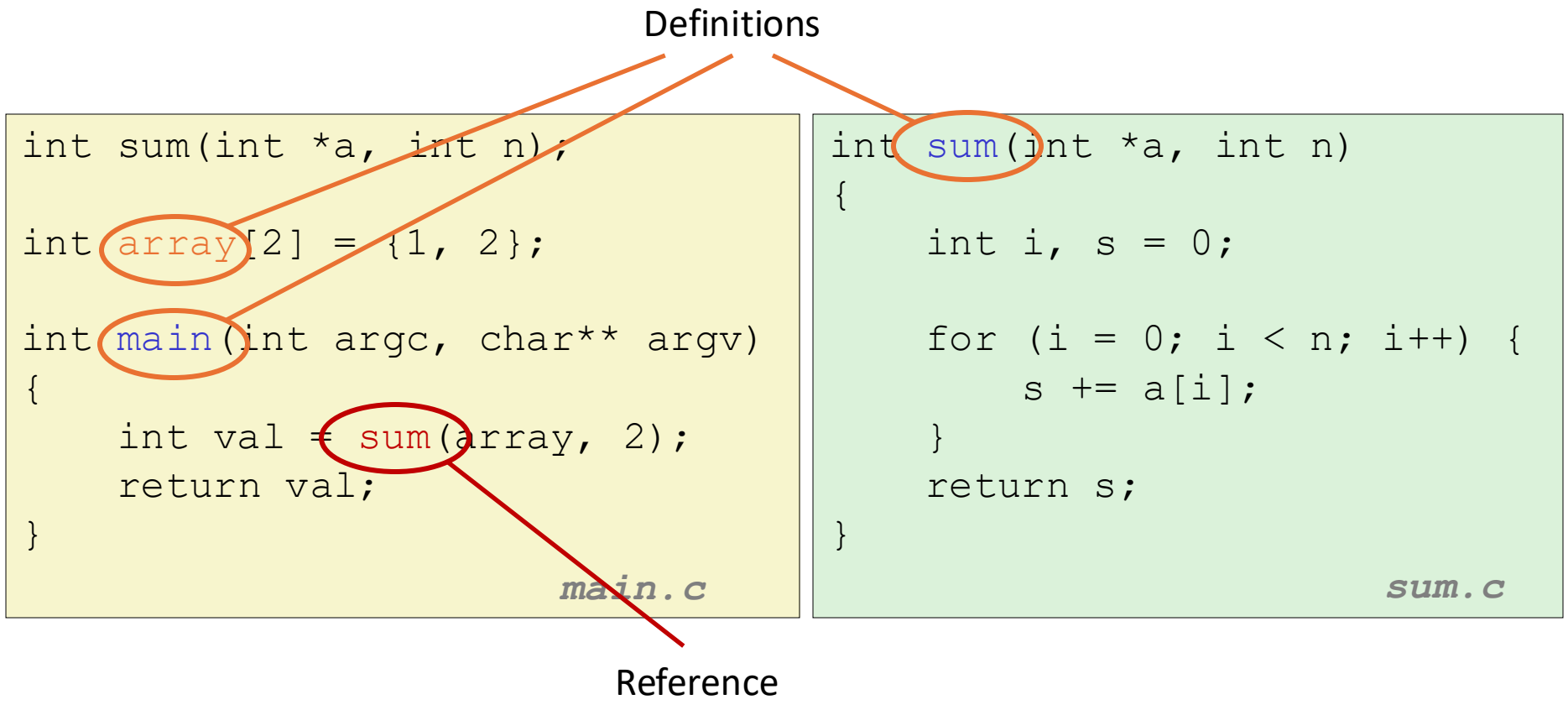
Symbol definitions are stored in object file in the **symbol table.**

- Symbol table is an array of entries
- Each table entry includes name, type, size, and location of symbol.

… three cases

- A symbol can be defined by the object file.
- It can be undefined; in which case the linker is required to find the definition and link the object file to the definition.
- It can be *multiply defined*. This is normally an error… but there are ways this can be done (we will not see these ways in this course)

Symbols in Example C Program

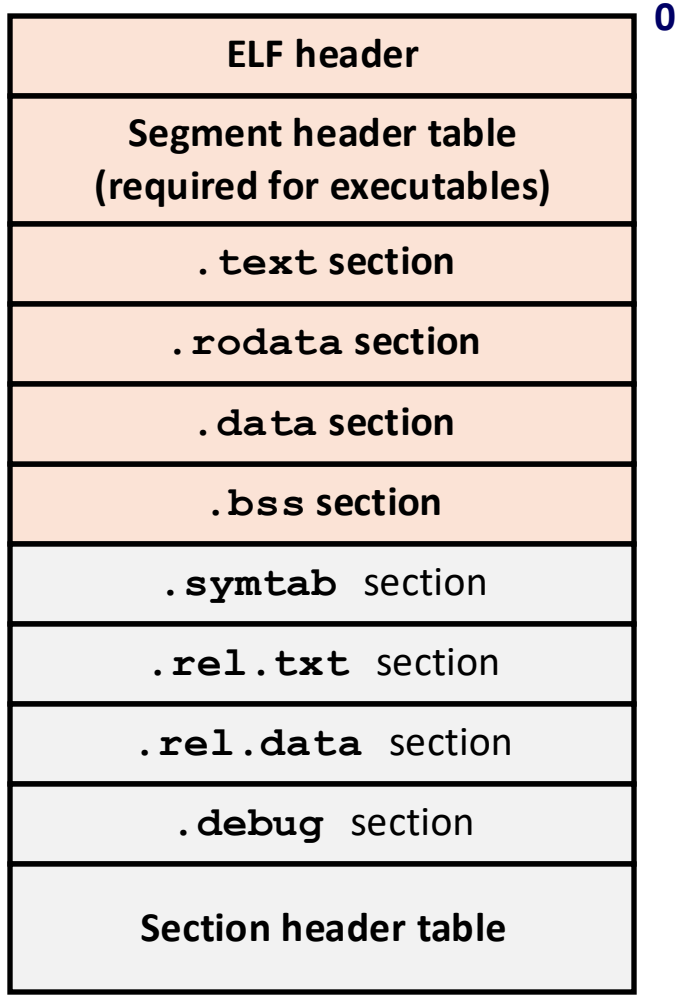


Linkers can “move things around”. We call this “relocation”

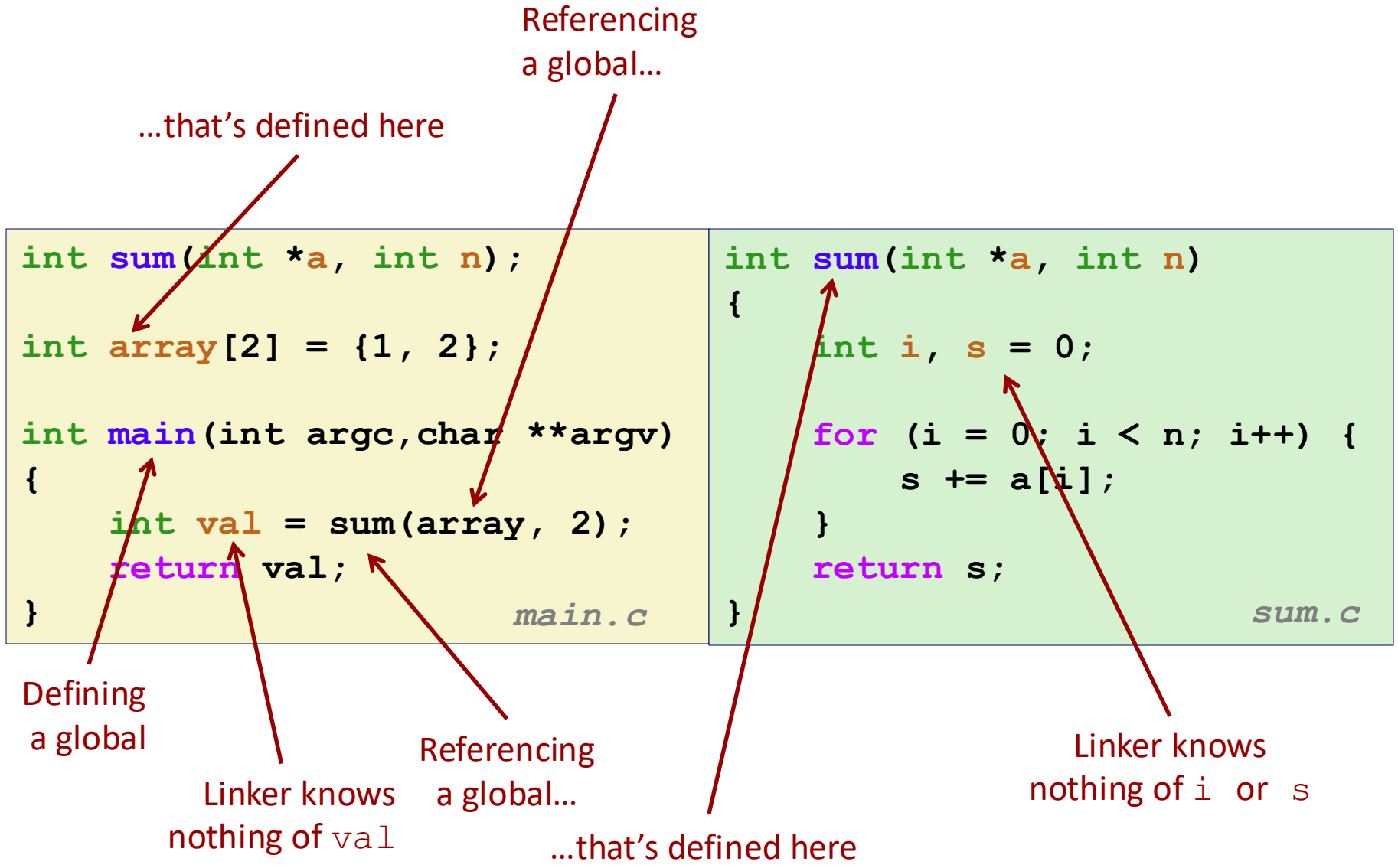
- A linker merges code and data sections into single sections
  - As part of this it *relocates* symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - It updates references to these symbols to reflect their new positions.

Object File Format (ELF)

- ELF header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual address memory segments + sizes.
- .text section (code)
- .rodata section (read-only data, jump offsets, strings)
- .data section (initialized global and static local variables)
- .bss section (name “bss” is lost in history)
  - Global or static local variables that weren’t initialized: zeros.
  - Has section header but occupies no space



Example of Symbol Resolution



Symbol Identification

Which of the following names will be in the symbol table of `symbols.o`?

symbols.**c**:

```
int incr = 1;
int foo(int a) {
  int b = a + incr;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

Names:

- incr
- foo
- a
- argc
- argv
- b
- main
- printf
- "%d\n"

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

Local Symbols

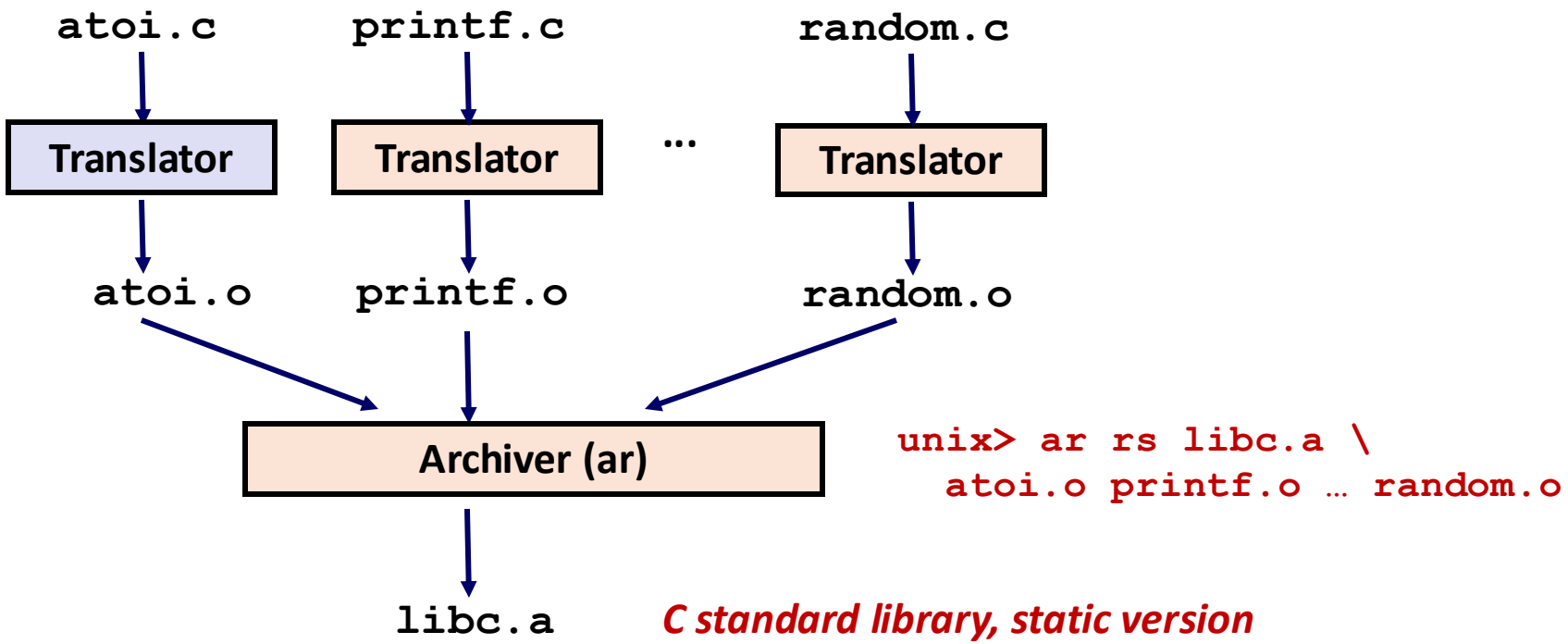
- Local non-static C variables vs. local static C variables
  - Local non-static C variables: stored on the stack
  - Local static C variables: stored in either `.bss` or `.data`

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.0` and `x.1`.

Static Libraries



- Archiver creates a single file that contains all the .o files, plus a lookup table (basically, a “directory”) that the linker can use to find the files.

Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar –t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar –t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries

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

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char**
argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
           z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

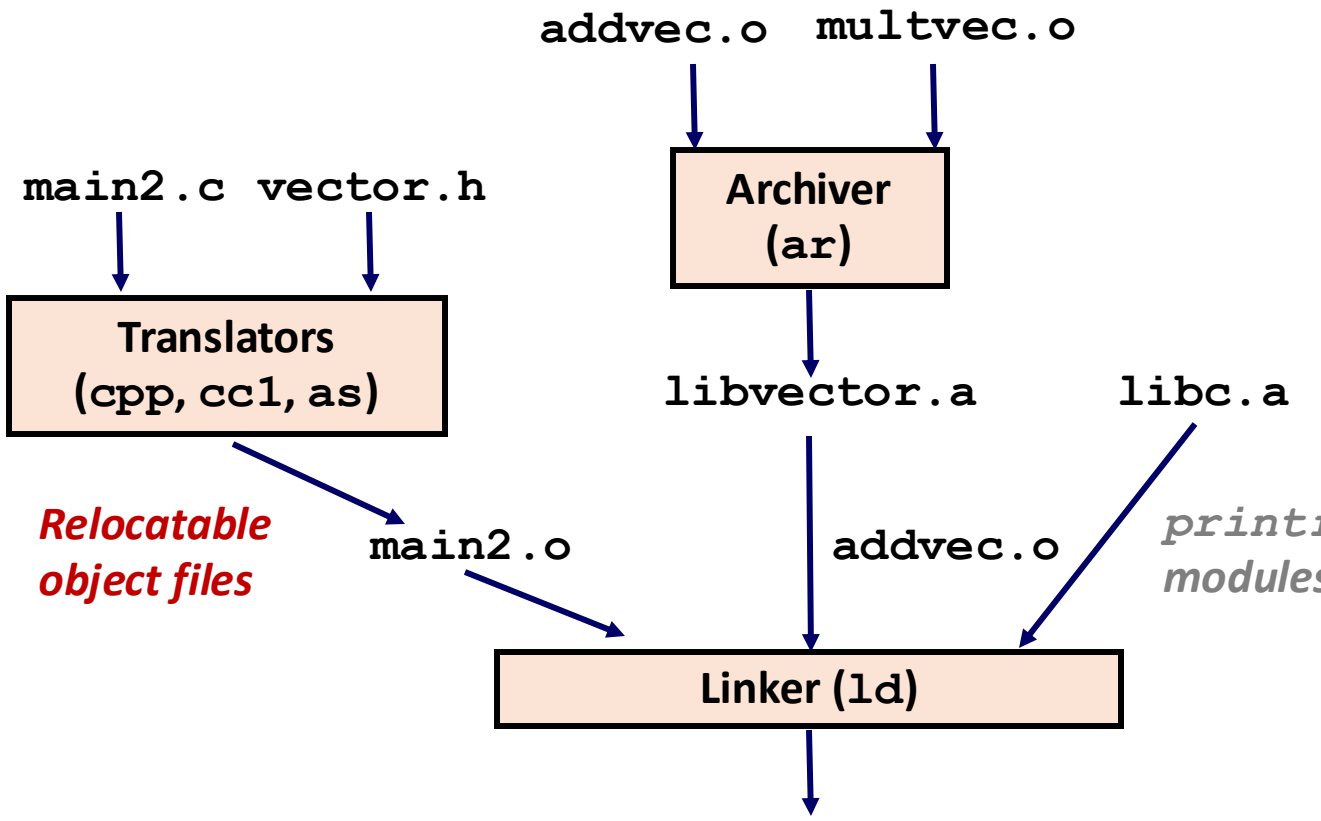
```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```c
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

Linking with Static Libraries




```
linux> gcc -static -o prog2c \
       main2.o -L. -lvector
```

New compiler options

- `-L`
  - To specify locations (directories) for library files
- `–l`
  - To specify the libraries to link with
- `-I`
  - To specify the locations (directories) for header files

Using Static Libraries

- Linker's algorithm for resolving external references:
  - Scan **`.o`** files and **`.a`** files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new **`.o`** or **`.a`** file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes in system libraries? Must rebuild everything!

Example: hugely disruptive 2016 library issue:

https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

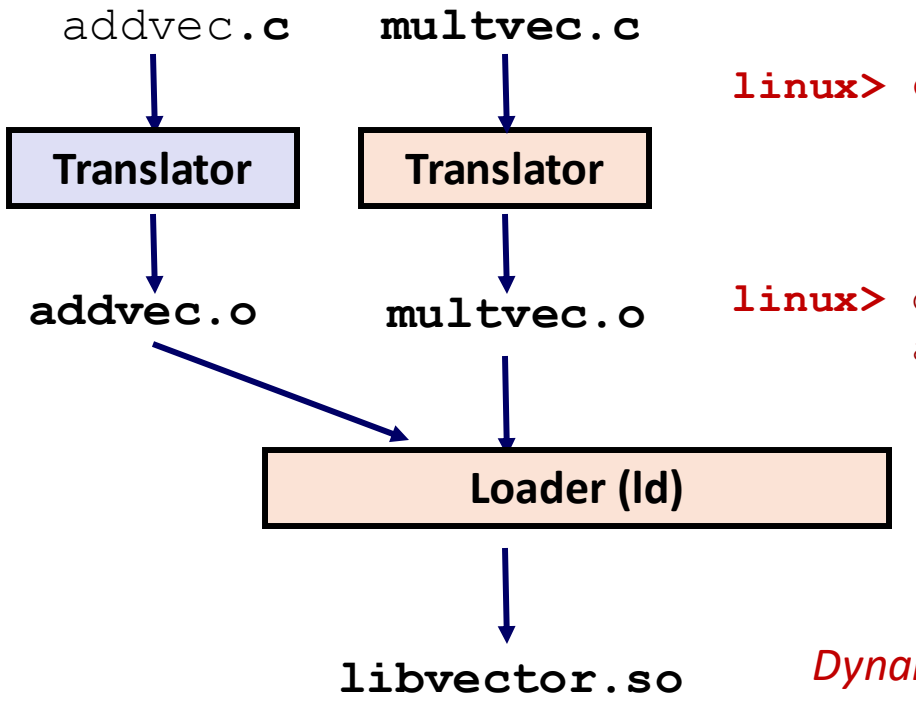
Shared Libraries

- Shared libraries save space and resolve this issue.

- Term refers to:
  - Object files that contain code and data.
  - Saved in a special directory (LOADPATH points to it).
  - Loaded and linked into an application dynamically, at either load-time or run-time
  - Also called: dynamic link libraries, DLLs, .so files

Dynamic Library Example

Tell the compiler to generate position independent code for shared libraries

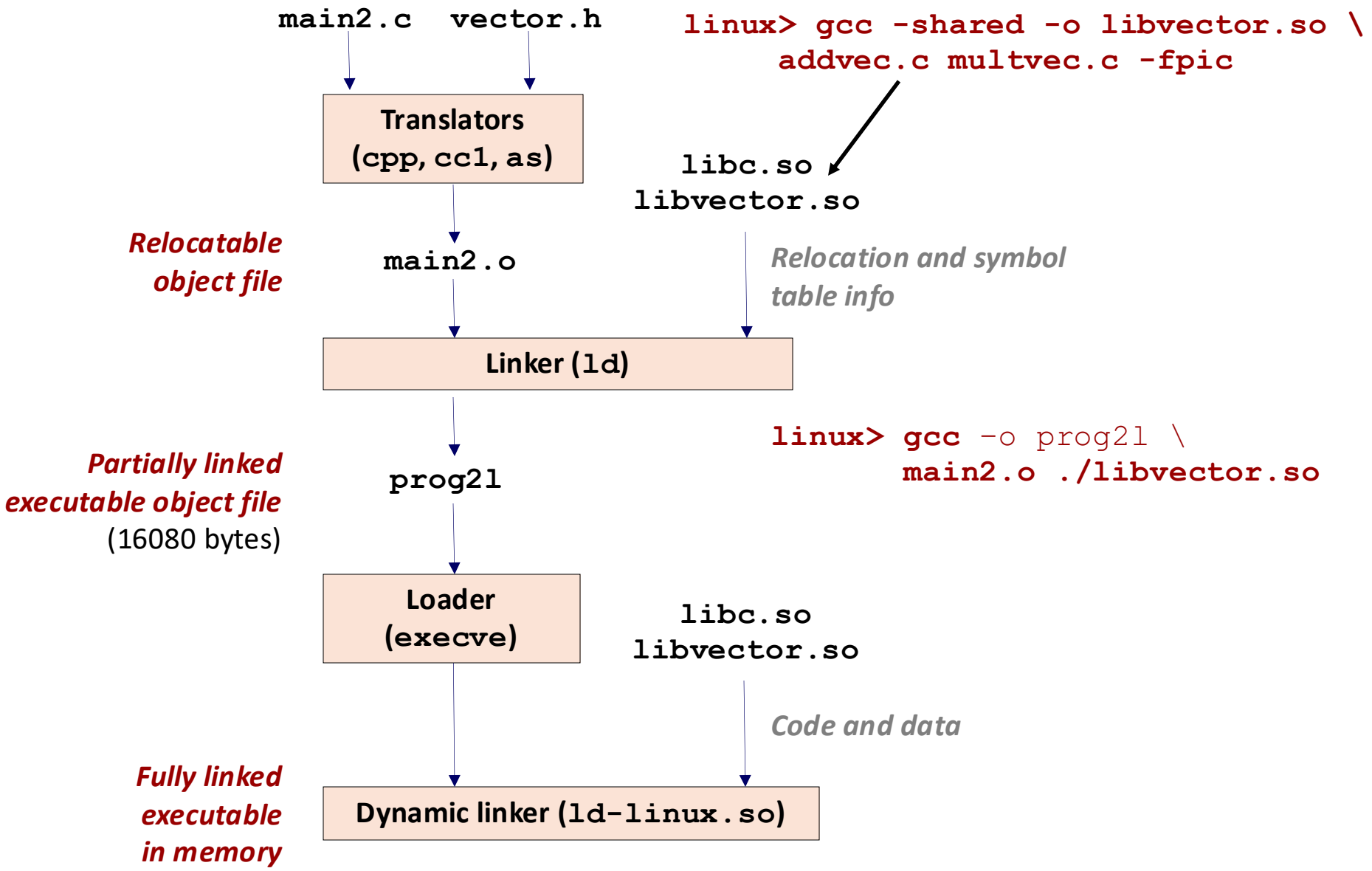



```
linux> gcc -Og -c addvec.c multvec.c -fpic
```

```
linux> gcc -shared -o libvector.so \
       addvec.o multvec.o
```

*Dynamic **vector library***

Dynamic Linking at Load-time



Runtime errors

- At runtime, your program searches for the .so file

- What if it can’t find it?
  - You will get an error message during execution, and the executable will terminate. Depending on the version of Linux, this occurs when you launch the program, or when it tries to access something in the dll

Some dll files also have “versioning” data. On these, your program might crash because of an “incompatible dll version number”

Linking Summary - I

- Linking is a technique that allows programs to be constructed from multiple object files

- Linking can happen at different times in a program's lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

- Understanding linking can help you avoid nasty errors and make you a better programmer

Linking Summary - II

- Usually: Just happens, no big deal
- But there are many sophisticated features and options!
- When using these fancier options, expect strange errors
  - Bad symbol resolution
  - Ordering dependence of linked .o, .a, and .so files
- For power users, it takes effort but then you can do:
  - *Interpositioning* to trace programs with & without source
  - We are not going to cover interpositioning in this course

Exercise: static linking

- Walk through the example tutorial and example at:
  - https://opensource.com/article/22/6/static-linking-linux

Makefiles

- With many source files to create multiple object files and many steps to complete, it would be nice to put all of these commands (compile instructions) into a script
  - That specialized compile script is called the Makefile
- We write *rules* in a Makefile with a simple format
- GNU make manual
  - https://www.gnu.org/software/make/manual/make.html
- A simple Makefile tutorial
  - https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

git and GitHub

- Let’s switch to the slides at:
  - https://courses.cs.washington.edu/courses/cse403/13au/lectures/git.ppt.pdf

Project 2

- Let’s clone the repository and copy it to our own private repo
  - Using the instructions at: https://github.com/CSCI210Mines/project2_spring25
  - Let’s analyze the Makefile and build an executable for Project 2