Linking

15-213: Introduction to Computer Systems
"13th" Lecture, July 3, 2019

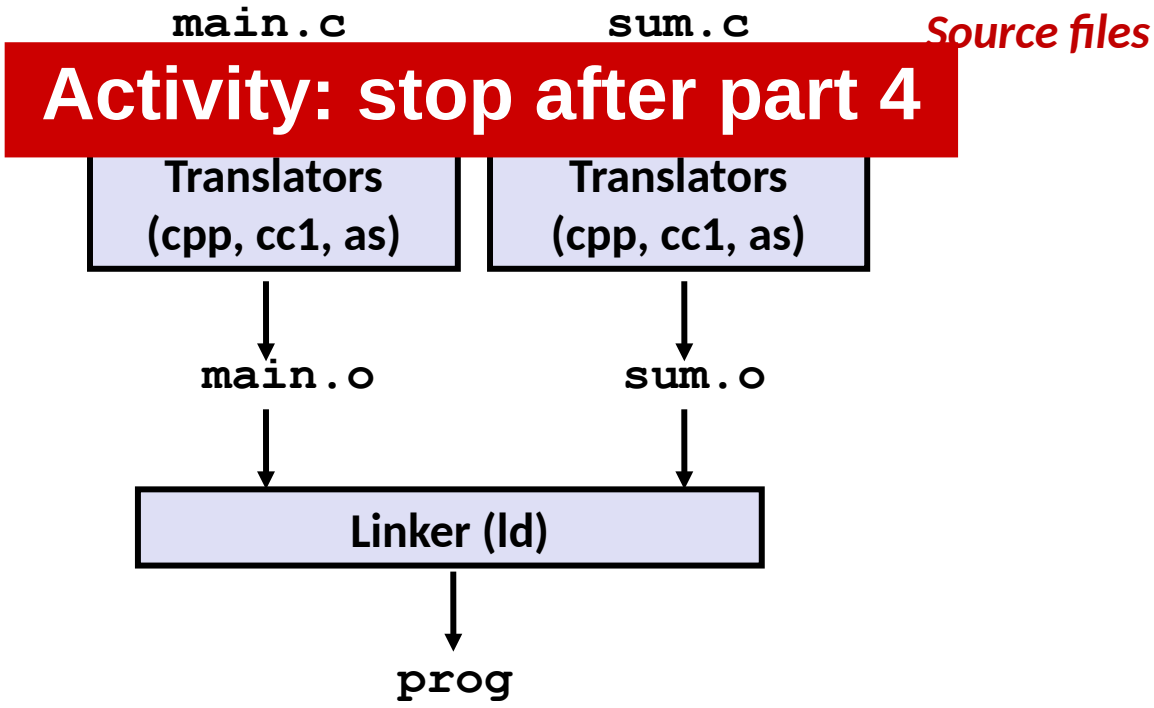
Instructor:
Sol Boucher

Today

- **Build Process**
  - **Translation**
  - Object files
- Linking
  - Motivation
  - How it works
  - Pitfalls
  - Libraries

The Build Process

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



The Preprocessor & The Role of .h Files

global.h

```
extern int g;
static int init = 0;
```

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

```
#else
  extern int g;
  static int init = 0;
#endif
```

c2.c

```
#define INITIALIZE
#include <stdio.h>
#include "global.h"

int main(int argc, char** argv) {
  if (init)
    // do something, e.g., g=31;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

```
int g = 23;
static int init = 1;
```

Compilation

- **Q: What does the C compiler produce?**
  - A: Assembly code for the target architecture
- **Q: What happens to the type annotations?**
  - A: They are erased during compilation!
- **Q: What does the assembler produce?**
  - …

Today

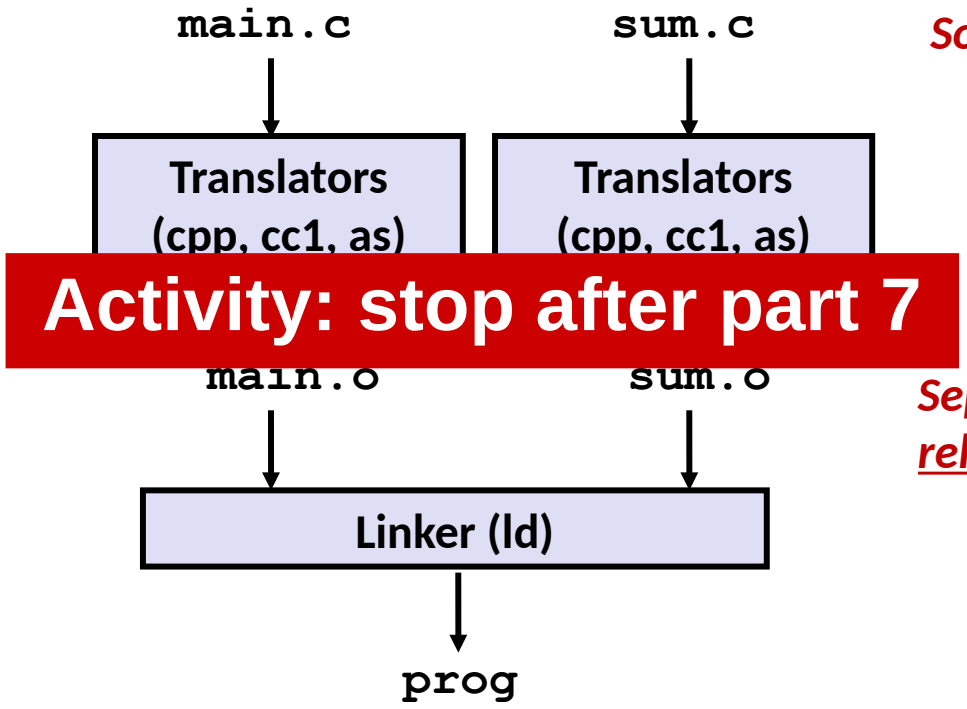
- **Build Process**
  - Translation
  - **Object files**
- Linking
  - Motivation
  - How it works
  - Pitfalls
  - Libraries

Three Kinds of Object Files (Modules)

- **Relocatable object file (`.o` file)**
  - Code and data in a form that can be combined with other relocatable object files
    - Each `.o` file is produced from exactly one source (`.c`) file
- **Executable object file (`a.out` file)**
  - Code and data that can be copied into memory and executed
- **Shared object file (`.so` file)**
  - Relocatable object that can be loaded into memory and linked dynamically, at either load time or run-time
  - Called *Dynamic Link Libraries* (DLLs) by Windows

The Build Process

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Source files

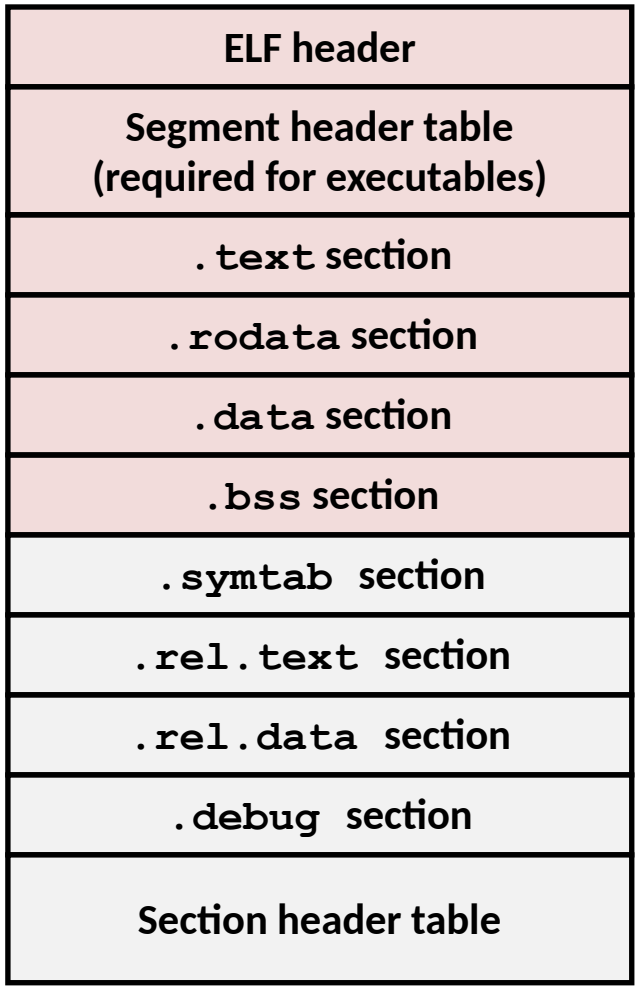
Separately compiled relocatable object files

ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space

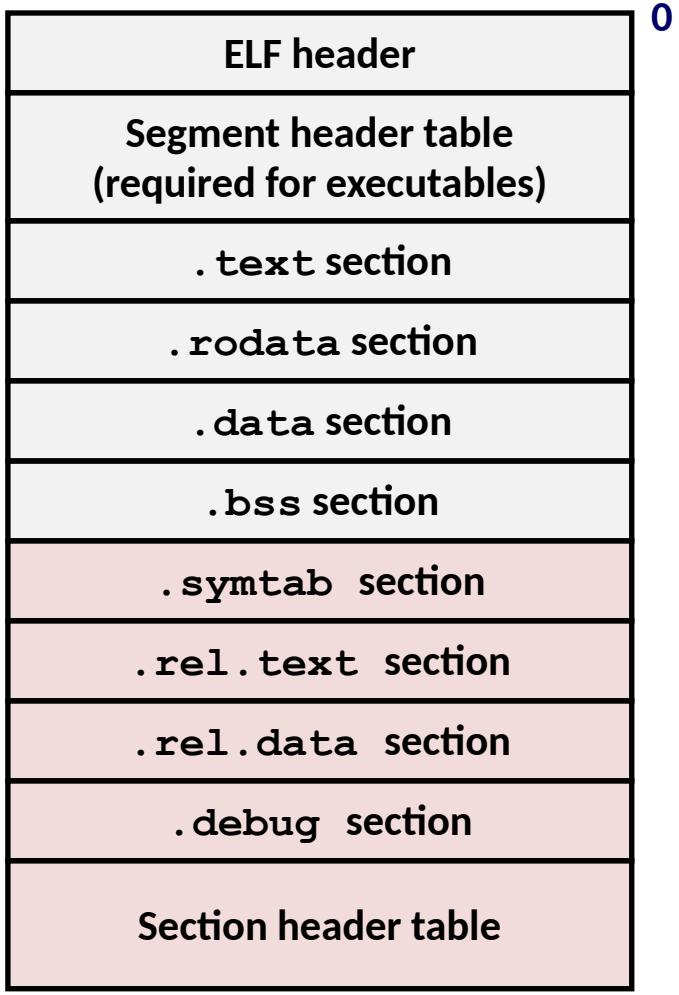
Executable!

Writable!



ELF Object File Format (cont.)

- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **`.rel.data` section**
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- **`.debug` section**
  - Info for symbolic debugging (`gcc -g`)
- **Section header table**
  - Offsets and sizes of each section



Relocation info

```
0000000000000013 <main>:
  13:   55                          push   %rbp
  14:   48 89 e5                    mov    %rsp,%rbp
      # printf("before: %d\n", global);
  17:   8b 05 00 00 00 00           mov    0x0(%rip),%eax # 1d
  1d:   89 c6                       mov    %eax,%esi
  1f:   48 8d 3d 00 00 00 00        lea    0x0(%rip),%rdi # 26
  26:   b8 00 00 00 00              mov    $0x0,%eax
  2b:   e8 00 00 00 00              callq  30 <main+0x1d>
  30:   bf 6d 3b 00 00              mov    $0x3b6d,%edi
      ...
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000019 R_X86_64_PC32     .data-0x0000000000000004
0000000000000022 R_X86_64_PC32     .rodata-0x0000000000000004
000000000000002c R_X86_64_PLT32    printf-0x0000000000000004
      ...
```

Today

- Build Process
  - Translation
  - Object files
- **Linking**
  - **Motivation**
  - How it works
  - Pitfalls
  - Libraries

Why Linkers?

- **Reason 1: Modularity**
  - Organize source code into multiple files
  - Link against separate existing library projects
- **Reason 2: Efficiency**
  - Time: Separate compilation
  - Space: Libraries

Today

- Build Process
  - Translation
  - Object files
- **Linking**
  - Motivation
  - **How it works**
  - Pitfalls
  - Libraries

What Do Linkers Do?

- **Step 1: 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 (by assembler) in *symbol table*.
  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

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;
}
```

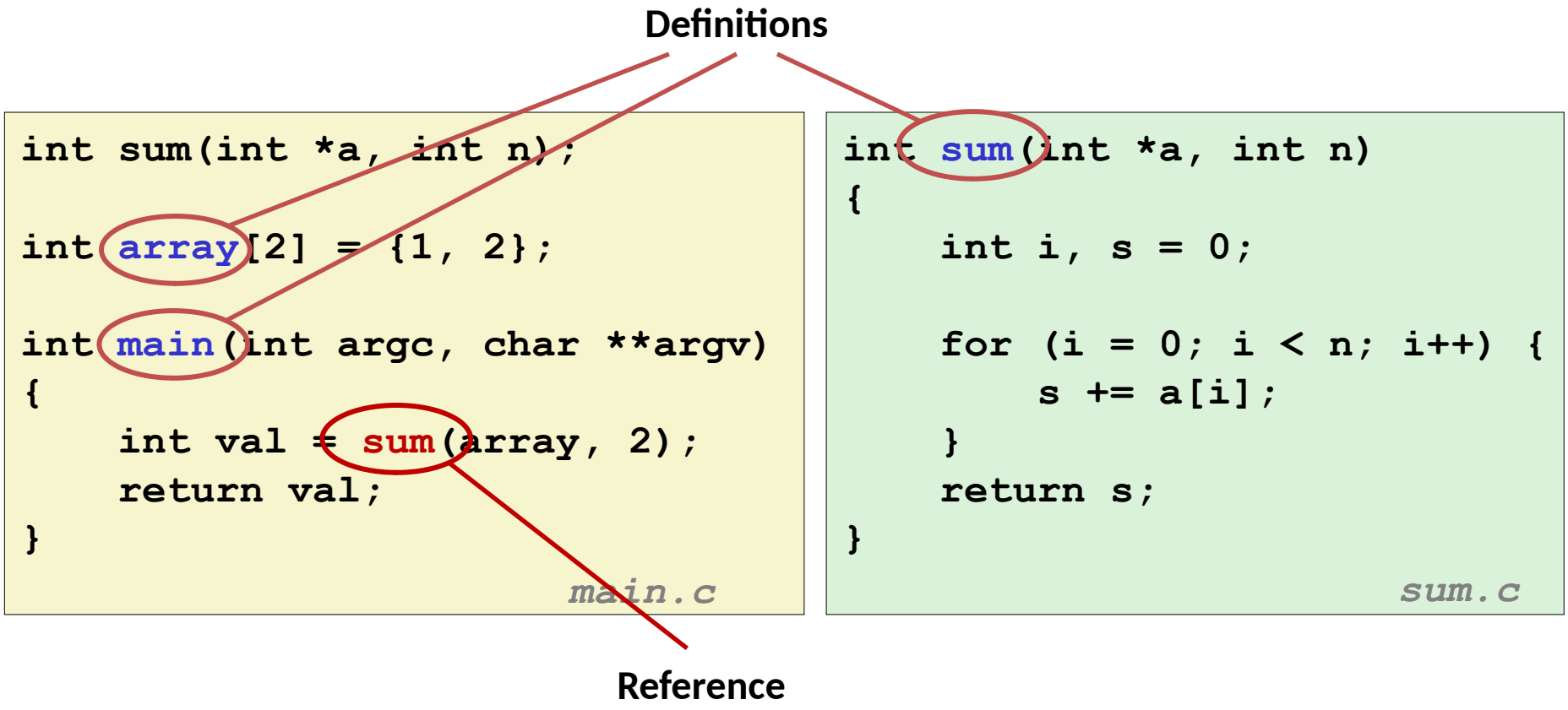
main.c

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

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

sum.c

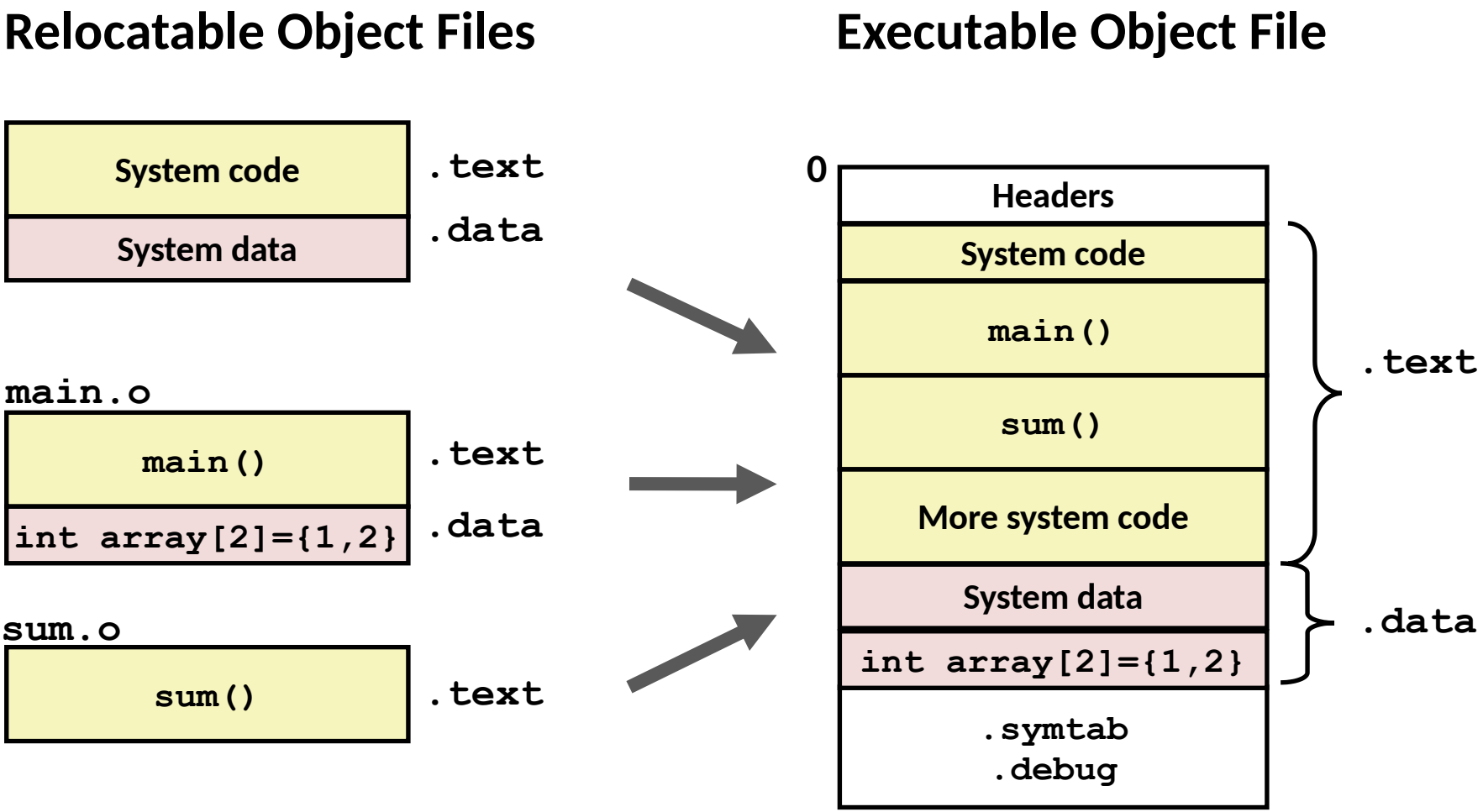
Symbols in Example C Program



What Do Linkers Do? (cont.)

- **Step 2: Relocation**
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Step 2: Relocation



Before and After Relocation

```
0000000000000013 <main>:
  13:  55                    push   %rbp
  14:  48 89 e5              mov    %rsp,%rbp
     # printf("before: %d\n", global);
  17:  8b 05 00 00 00 00     mov    0x0(%rip),%eax # 1d
  1d:  89 c6                 mov    %eax,%esi
  1f:  48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi # 26
  26:  b8 00 00 00 00        mov    $0x0,%eax
  2b:  e8 00 00 00 00        callq  30 <main+0x1d>
  30:  bf 6d 3b 00 00        mov    $0x3b6d,%edi
     ...                                              main.o
```

```
0000000000001148 <main>:
1148:  55                    push   %rbp
1149:  48 89 e5              mov    %rsp,%rbp
114c:  8b 05 de 2e 00 00     mov    0x2ede(%rip),%eax # 4030 <global>
1152:  89 c6                 mov    %eax,%esi
1154:  48 8d 3d a9 0e 00 00  lea    0xea9(%rip),%rdi # 2004
115b:  b8 00 00 00 00        mov    $0x0,%eax
1160:  e8 cb fe ff ff        callq  1030 <printf@plt>
1165:  bf 6d 3b 00 00        mov    $0x3b6d,%edi
     ...                                              main
```

Linker Symbols

- **Global symbols**
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-**`static`** C functions and non-**`static`** global variables.

- **External symbols**
  - Global symbols that are referenced by module *m* but defined by some other module.

- **Local symbols**
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: C functions and global variables defined with the **`static`** attribute.
  - **Local linker symbols are *not* local program variables**

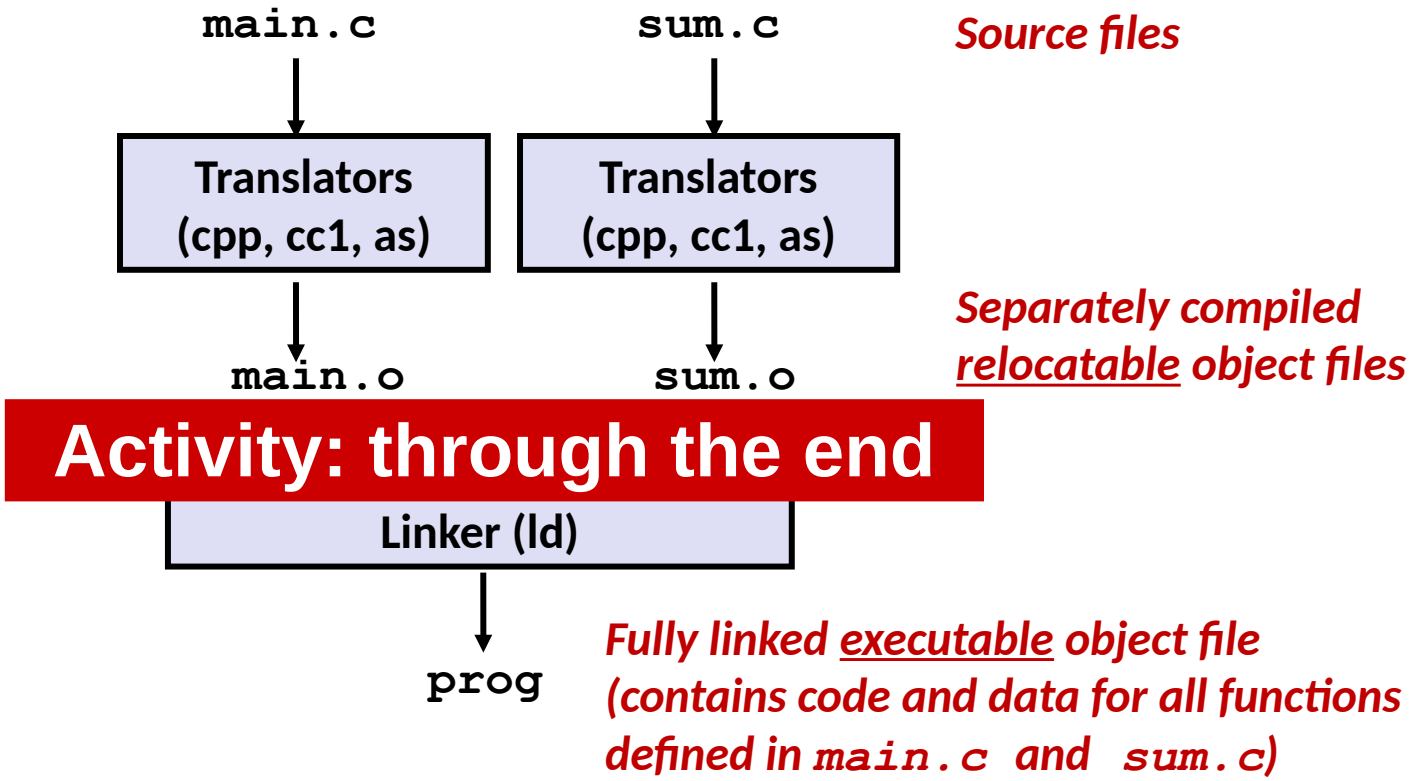
Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable

The Build Process

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Today

- Build Process
  - Translation
  - Object files
- **Linking**
  - Motivation
  - How it works
  - **Pitfalls**
  - Libraries

Pitfall: Duplicate Symbols

```c
int global = 0;

int main(void)
{
    set_global(15213);
    return 0;
}
```

main_zero.c

```c
int global = 0;

void set_global(int val)
{
    global = val;
}
```

helper.c

```
$ gcc -c helper.o
$ gcc -c main_zero.o
$ gcc -o main_zero main_zero.o helper.o
/usr/bin/ld: helper.o:(.bss+0x0): multiple definition of `global'
; main_zero.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
$
```

Pitfall: Duplicate Symbols

weak symbol
- Uninitialized globals
- Linker allows multiple, picks arbitrary one

strong symbol
- Procedures, initialized globals
- Linker allows only one

```
int global;

int main(void)
{
    set_global(15213);
    return 0;
}
```
main_zero.c

```
int global = 0;

void set_global(int val)
{
    global = val;
}
```
helper.c

```
$ gcc -c helper.o
$ gcc -c main_zero.o
$ gcc -o main_zero main_zero.o helper.o
$ ./main_zero
15213
$
```

Pitfall: Clashing Types

```
float global;

int main(void)
{
    set_global(15213.0);
    printf("%f\n", global);
    return 0;
}
```

%xmm0

main_scary.c

```
int global = 0;

void set_global(int val)
{
    global = val;
}
```

%edi

helper.c

```
$ gcc -c helper.o
$ gcc -c main_scary.o
$ gcc -o main_scary main_scary.o helper.o
$ ./main_scary
0.000000
$
```

Pitfall: Clashing Qualifiers

.data? .bss?

```
int global;

int main(void)
{
    global = 15213;
    return 0;
}
```

main_scary.c

.rodata!

```
const int global = 0;
```

helper.c

```
$ gcc -c helper.o
$ gcc -c main_zero.o
$ gcc -o main_zero main_zero.o helper.o
$ ./main_scary
Segmentation fault
$
```

Takeaway: Declare in Common Header

```c
#include "helper.h"

int main(void)
{
    set_global(15213);
    printf("%d\n", global);
    return 0;
}
```

main_scary.c

compiler error if types mismatched!

```c
#include "helper.h"

int global = 0;

void set_global(int val)
{
    global = val;
}
```

helper.c

linker error if forgotten!

```c
#ifndef HELPER_H_
#define HELPER_H_

extern int global;
void set_global(int);

#endif
```

helper.h

`extern` forces to be a *declaration*, not a weak definition

function prototypes are `extern` by default

Today

- Build Process
  - Translation
  - Object files
- **Linking**
  - Motivation
  - How it works
  - Pitfalls
  - **Libraries**

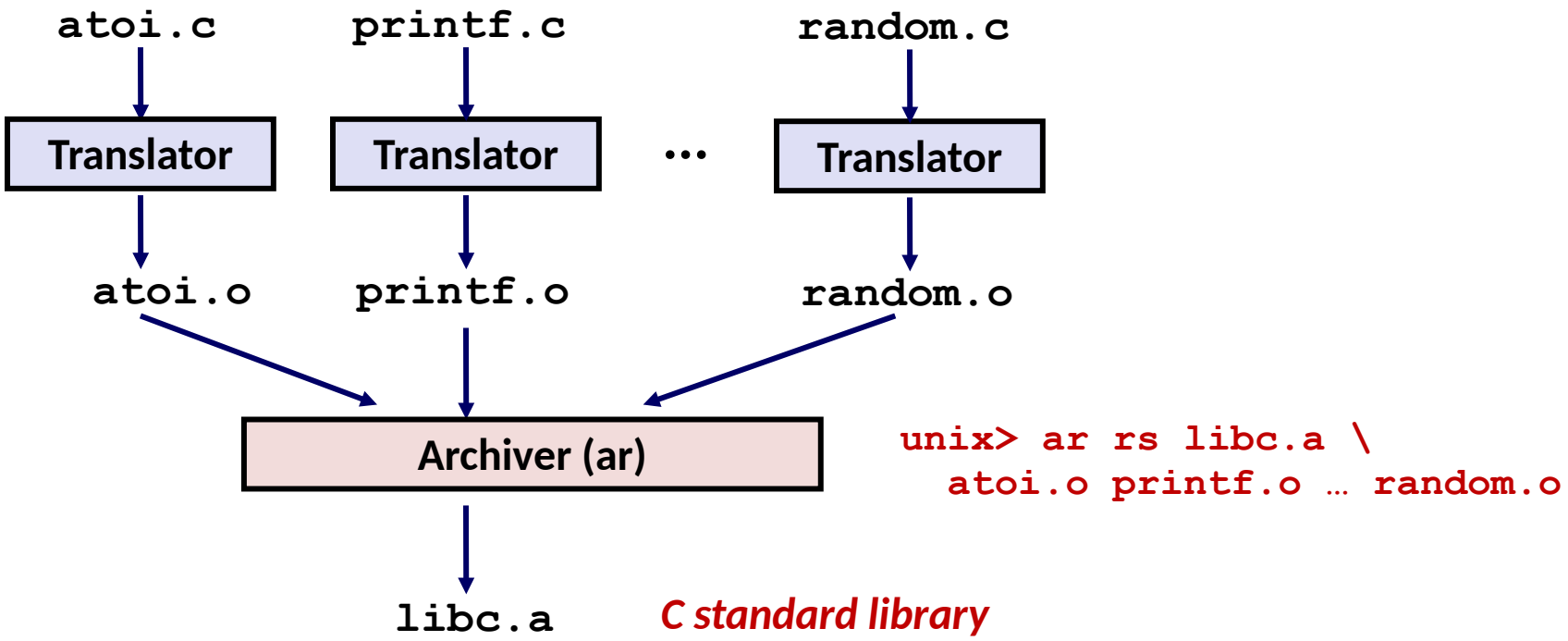
Quiz

Packaging Commonly Used Functions

- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

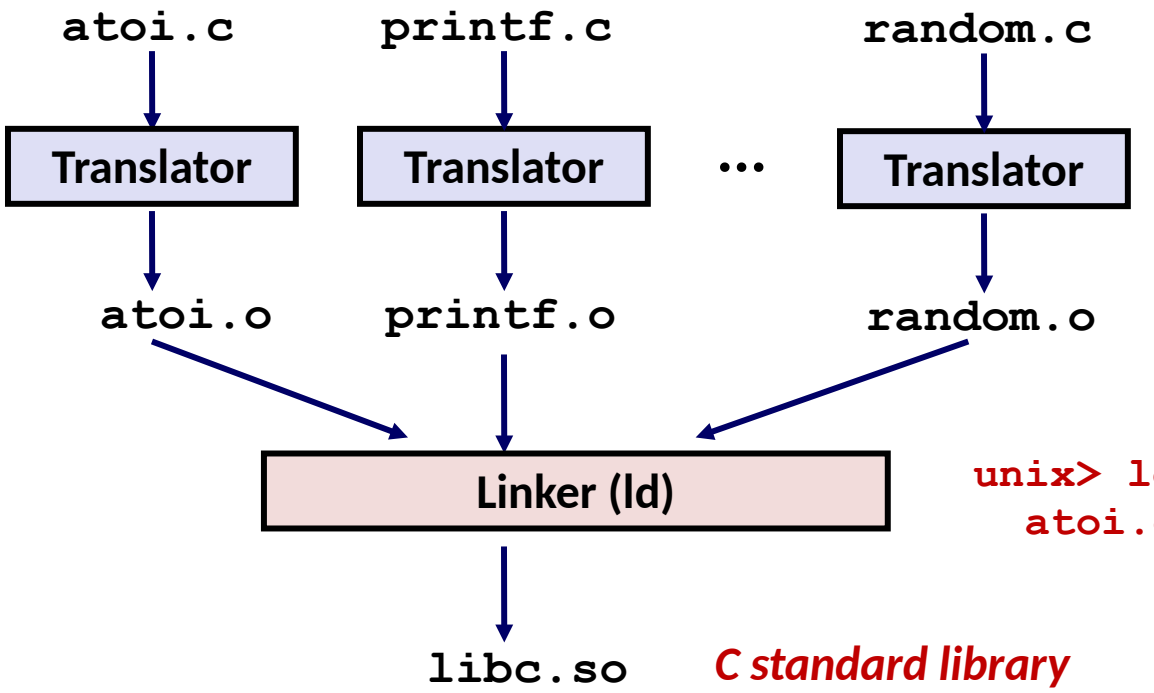
- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Static Libraries: Just an Archive of .o Files!



- Archiver allows incremental updates:
  - Recompile function that changes and replace .o file in archive.
- But then have to *recompile every executable on the system*!!!!1

Shared Libraries: Loadable at *Load* Time!




```
unix> ld -shared -o libc.so \
  atoi.o printf.o … random.o
```

- Cannot incrementally update the *library*:
  - Instead, must relink entire dynamic shared object.
- But replacing the library *automatically "updates" every executable*!

Remember this Linked Code?

```
0000000000001148 <main>:
1148:  55                      push   %rbp
1149:  48 89 e5                mov    %rsp,%rbp
114c:  8b 05 de 2e 00 00       mov    0x2ede(%rip),%eax # 4030 <global>
1152:  89 c6                   mov    %eax,%esi
1154:  48 8d 3d a9 0e 00 00    lea    0xea9(%rip),%rdi # 2004
115b:  b8 00 00 00 00          mov    $0x0,%eax
1160:  e8 cb fe ff ff          callq  1030 <printf@plt>
1165:  bf 6d 3b 00 00          mov    $0x3b6d,%edi
      ...
```

main

```
Disassembly of section .plt:
0000000000001020 <.plt>:
1020:  ff 35 e2 2f 00 00  pushq  0x2fe2(%rip) # 4008 <_GLOBAL_OFFSET_TAB
1026:  ff 25 e4 2f 00 00  jmpq   *0x2fe4(%rip) # 4010 <_GLOBAL_OFFSET_TA
102c:  0f 1f 40 00        nopl   0x0(%rax)

0000000000001030 <printf@plt>:
1030:  ff 25 e2 2f 00 00  jmpq   *0x2fe2(%rip) # 4018 <printf@GLIBC_2.2.
1036:  68 00 00 00 00     pushq  $0x0
103b:  e9 e0 ff ff ff     jmpq   1020 <.plt>
```

main

Linking Summary

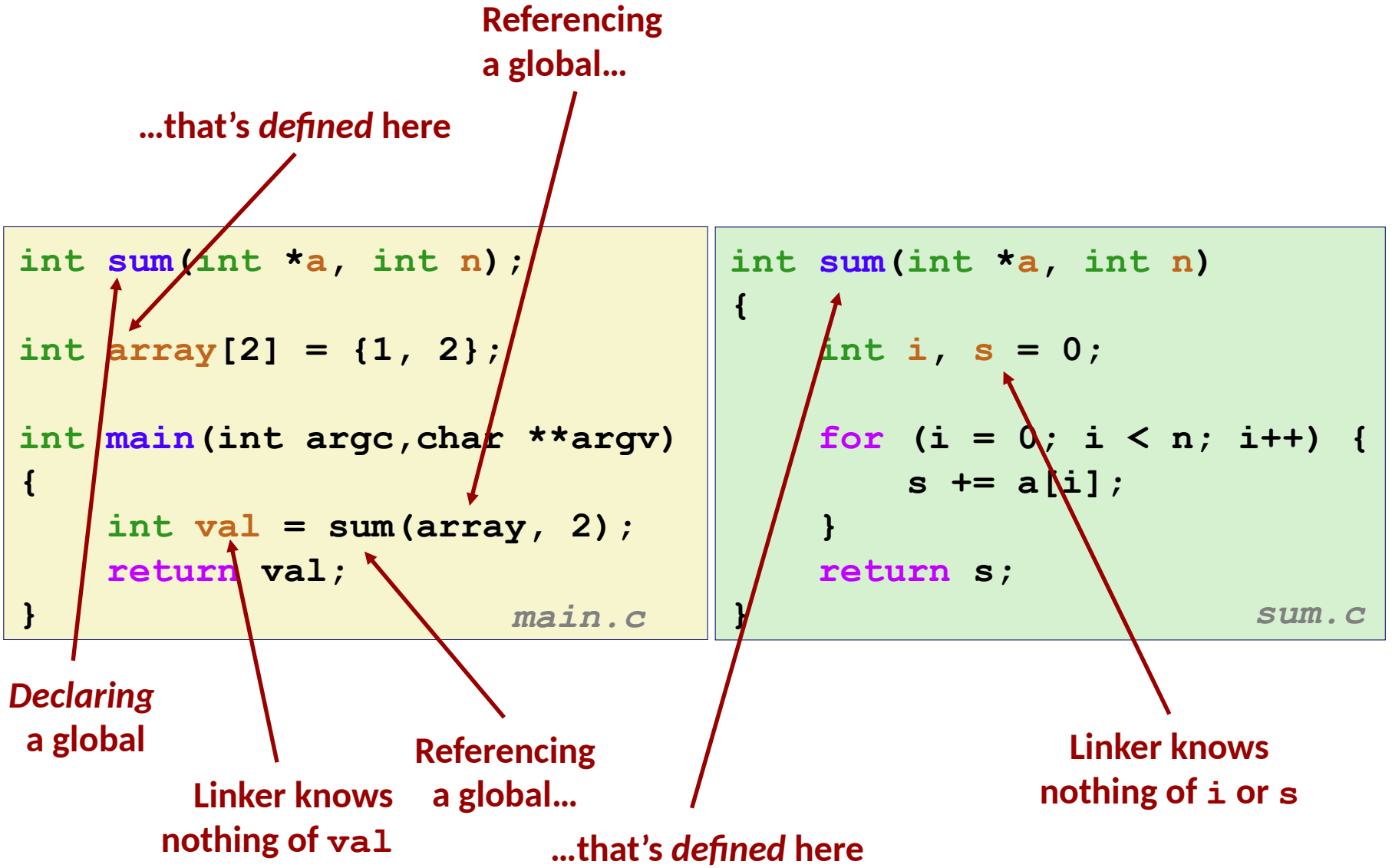
- **Linking is a technique that allows programs to be constructed from multiple object files.**

- **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) [`man dlopen` for more]

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

Appendix

Symbol Resolution



Symbol Identification

How many of the following names will be in the symbol table of `main.o`?

main.c:

```
int time;

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

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

Names:

- `time`
- `foo`
- `a`
- `b`
- `main`
- `argc`
- `argv`
- `printf`

From Sat Garcia, U. San Diego, used with permission

Symbol Identification

How many of the following names will be in the symbol table of `main.o`?

main.c:

```
int time;

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

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

Names:

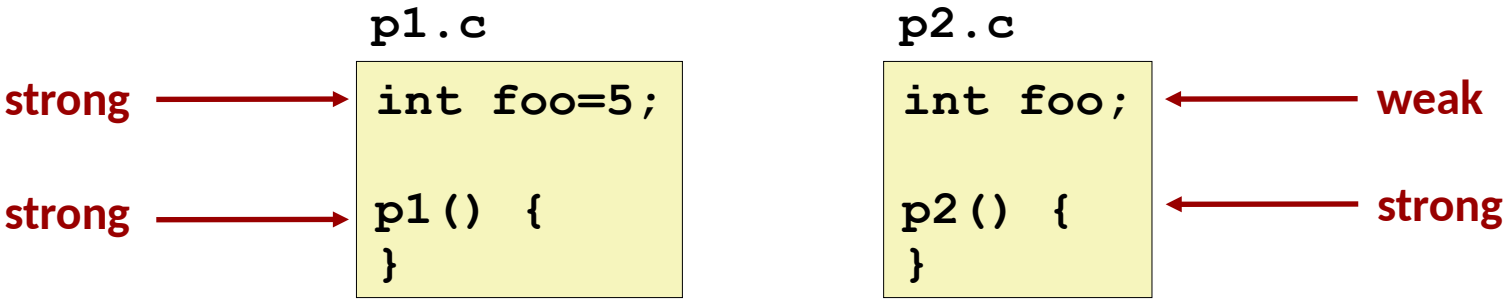
- time
- foo
- a
- b
- main
- argc
- argv
- printf

4

From Sat Garcia, U. San Diego, used with permission

How Linker Resolves Duplicate Symbol Definitions

- **Program symbols are either *strong* or *weak***
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals



Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`
- **Puzzles on the next slide**

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (**p1**)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** will overwrite **y**! Nasty!

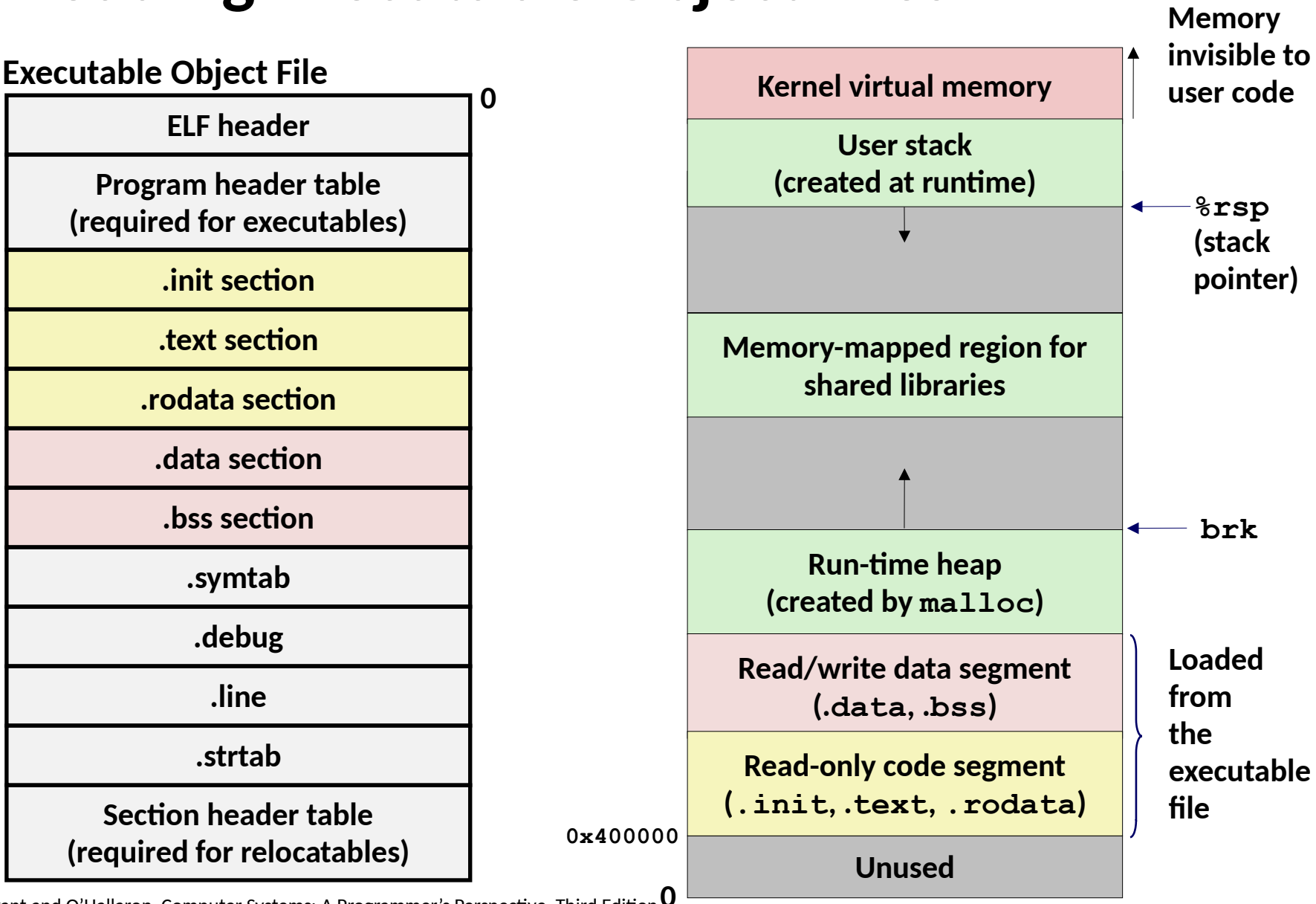
```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Loading Executable Object Files



Old-fashioned Solution: Static Libraries

- **Static libraries (`.a` archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

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 libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t 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

```
#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

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

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

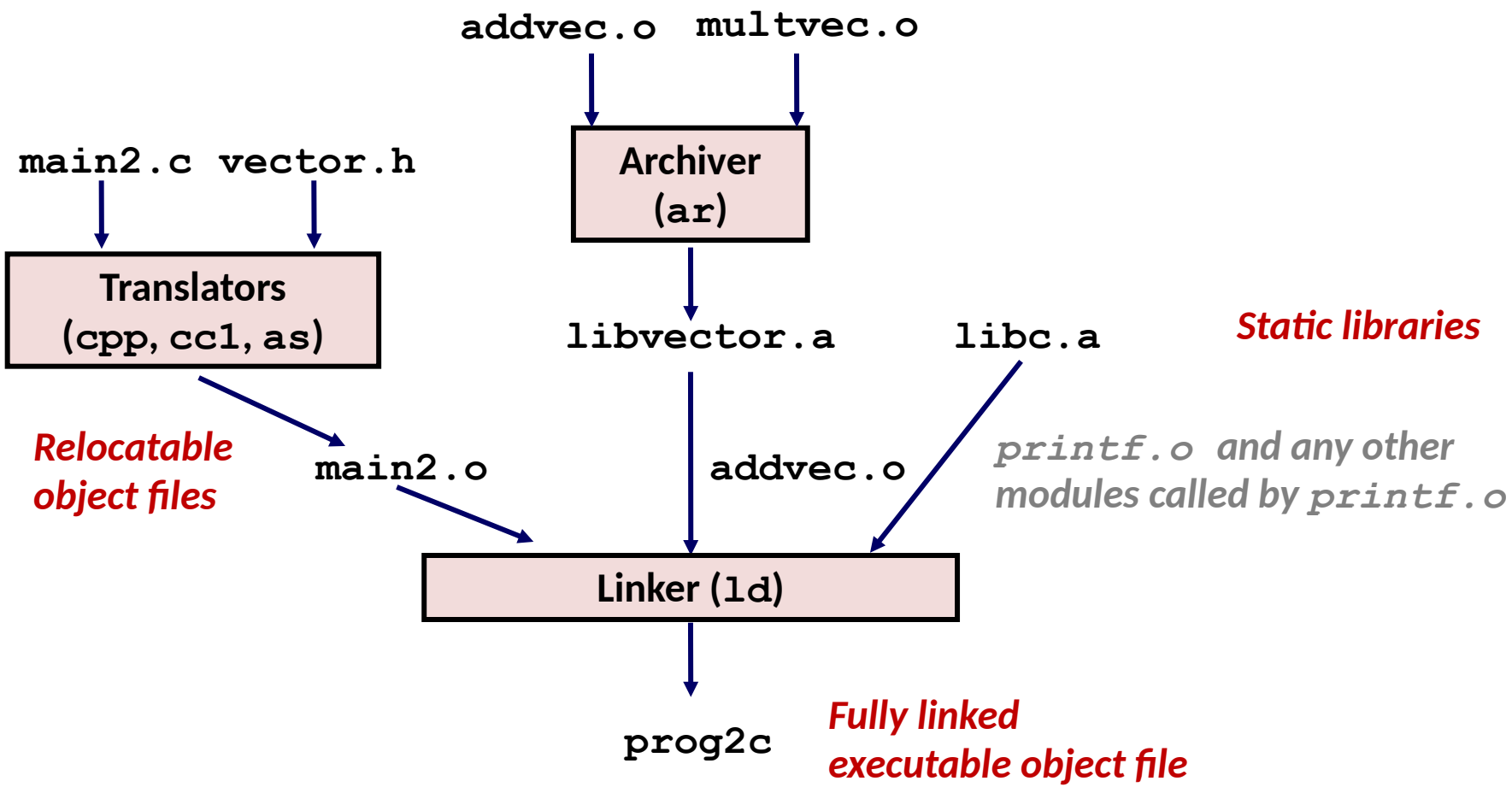
addvec.c

```
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



"c" for "compile-time"

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 -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function 'main':
libtest.o(.text+0x4): undefined reference to 'libfun'
```

Modern Solution: 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 of system libraries require each application to explicitly relink
    - Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- **Modern solution: Shared Libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically*, at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

Shared Libraries (cont.)

- **Dynamic linking can occur when executable is first loaded and run (load-time linking).**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.
- **Dynamic linking can also occur after program has begun (run-time linking).**
  - In Linux, this is done by calls to the `dlopen()` interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.
- **Shared library routines can be shared by multiple processes.**
  - More on this when we learn about virtual

What dynamic libraries are required?

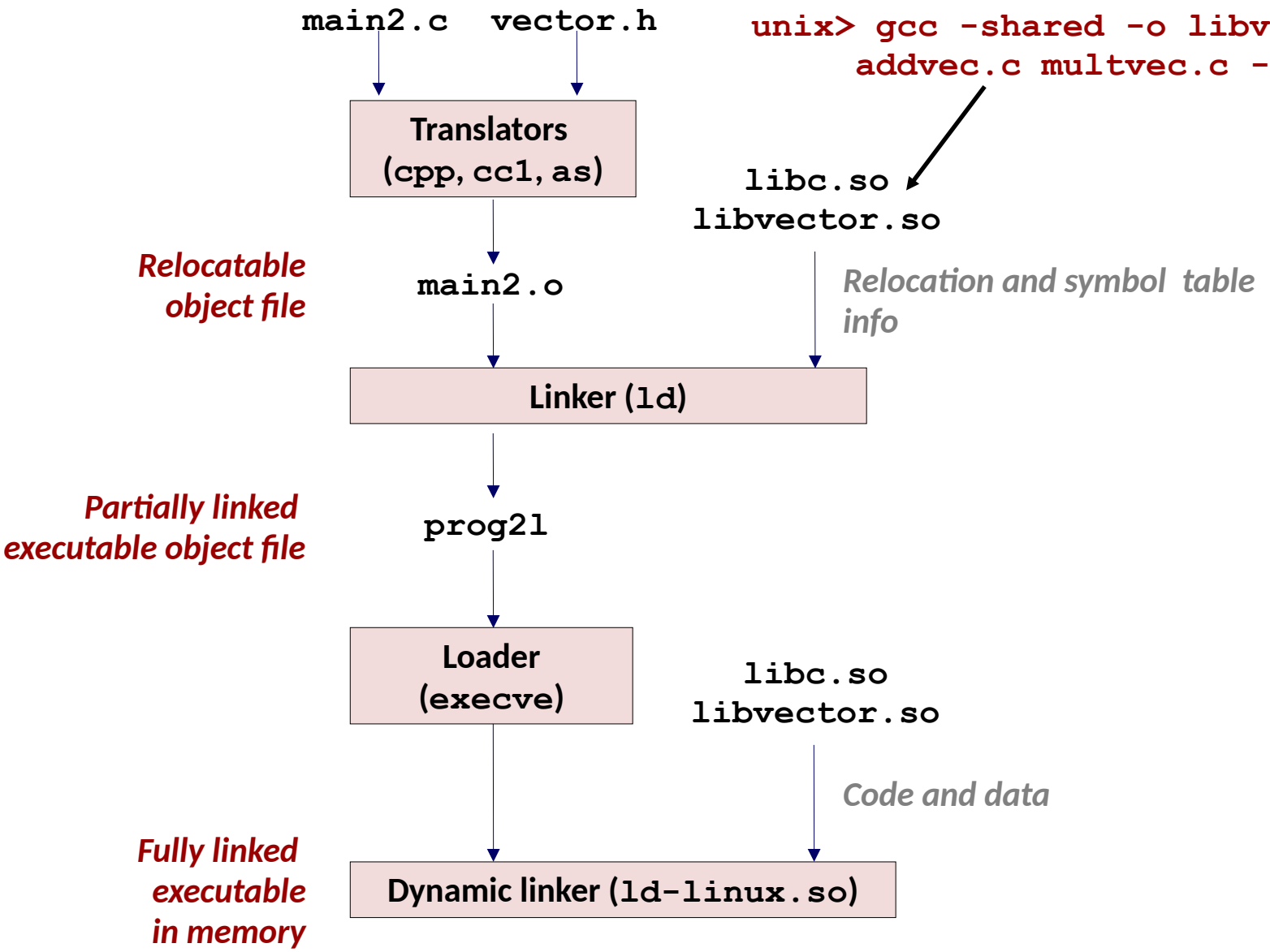
- **.interp section**
  - Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)
- **.dynamic section**
  - Specifies the names, etc of the dynamic libraries to use
  - Follow an example of csim-ref from cachelab

```
(NEEDED)                Shared library: [libm.so.6]
```

- **Where are the libraries found?**
  - Use "**`ldd`**

```
unix> ldd csim-ref
  linux-vdso.so.1 =>  (0x00007ffc195f5000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f345eda6000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f345f181000)
```

Dynamic Linking at Load-time



Dynamic Linking at Run-time

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

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

int main(int argc, char** argv)
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
  . . .
```

dll.c

Dynamic Linking at Run-time (cont)

```
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Dynamic Linking at Run-time