LINKING… HOW BASIC MECHANISMS ENABLE SOPHISTICATED WRAPPERS

Professor Ken Birman

CS4414 Lecture 12

SYSTEMS PROGRAMMING IS ABOUT TAKING CONTROL OVER EVERYTHING

We have seen that a systems programmer learns to “program” the hardware, operating system and software, including the C++ compiler itself, which we “program” via templates.

Today we will look at how linking works, and by doing so, we will discover another obscure example of a programmable feature that you might not normally expect to be able to control!

CORE SCENARIO

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?

IDEA MAP FOR TODAY

Libraries

Compiling to an object file

Static versus dynamic linking in Linux.

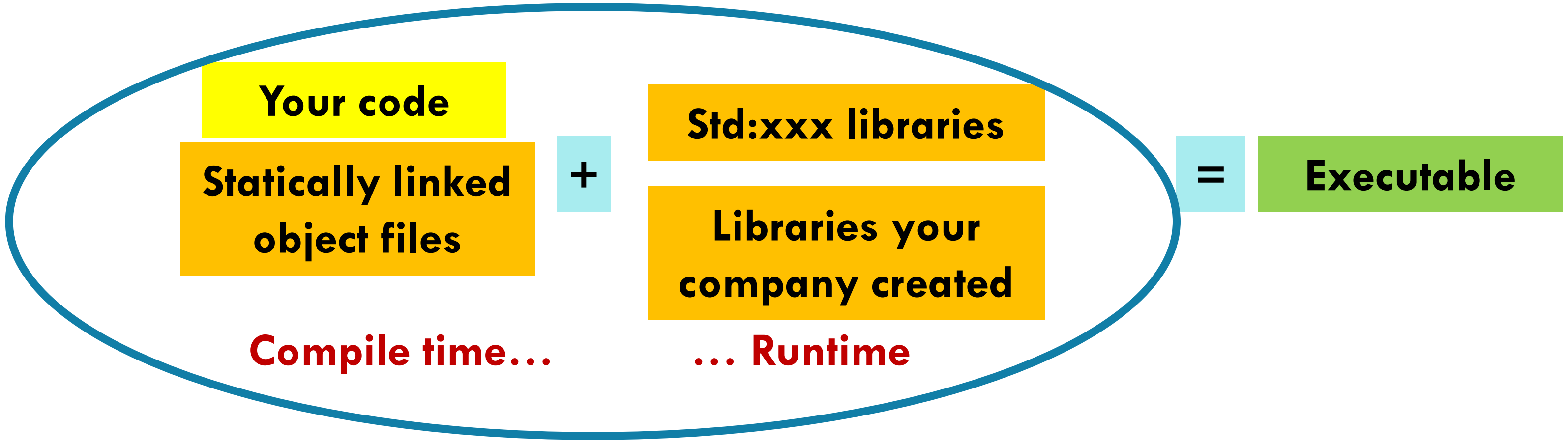
Main part of lecture. Be sure to understand this.

Dynamic linking: -shared -fPIC compilation. DLL segments, issue of base address

Wrappers for method interpositioning: a "super hacker" technique!

Insane/weird part, introduces some amazing features

LINKING


Your code
Statically linked object files
+
Std:xxx libraries
Libraries your company created
=
Executable
Compile time…
… Runtime

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++ IS THE SAME)

```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

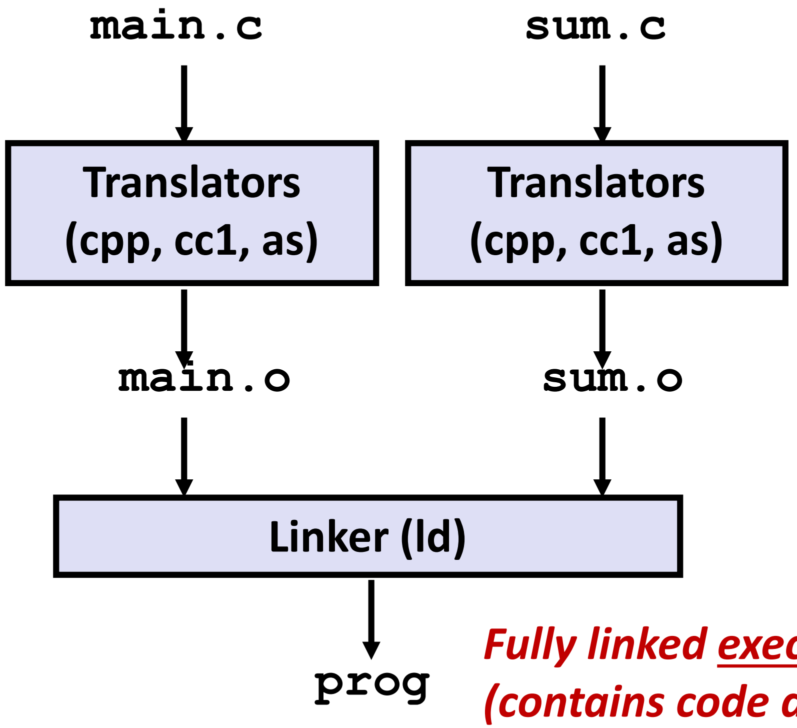
```c
int sum(int *a, int n)
{
    int i, s = 0;

    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
```



main.c

sum.c

Translators (cpp, cc1, as)

Translators (cpp, cc1, as)

main.o

sum.o

Linker (ld)

prog

Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in `main.c` *and* `sum.c`*)*

WHY LINKERS? REASON 1: 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++ class normally has its own hpp file (declares the type signatures of the methods and fields) and a separate cpp file (implements the class).

For fancy templated classes, C++ itself creates the needed cpp files, one for each distinct type-parameters list.

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.

REASON 2: LIBRARIES

Libraries aggregate common functions or classes.

Static linking combines modules of a program, but also used to be the main way of linking to libraries:

- Executables include copies of any library modules they reference (but just those .o files, not others in the library)
- Executable is complete and self-sufficient. It should run on any machine with a compatible architecture.

REASON 2: LIBRARIES

Dynamic linking is more common today

- Your executable program doesn't need to contain library code
- At execution, single copy of library code is shared, but the dynamic linker does need to be able to find the library file (a ".so" file)

If a dynamically linked executable is launched on a machine that lacks the DLL, you will get an error message (usually, on startup, but there are some obscure cases where it happens later, when the DLL is needed)

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.
- With C++ the “location” is the “namespace” that declared the class

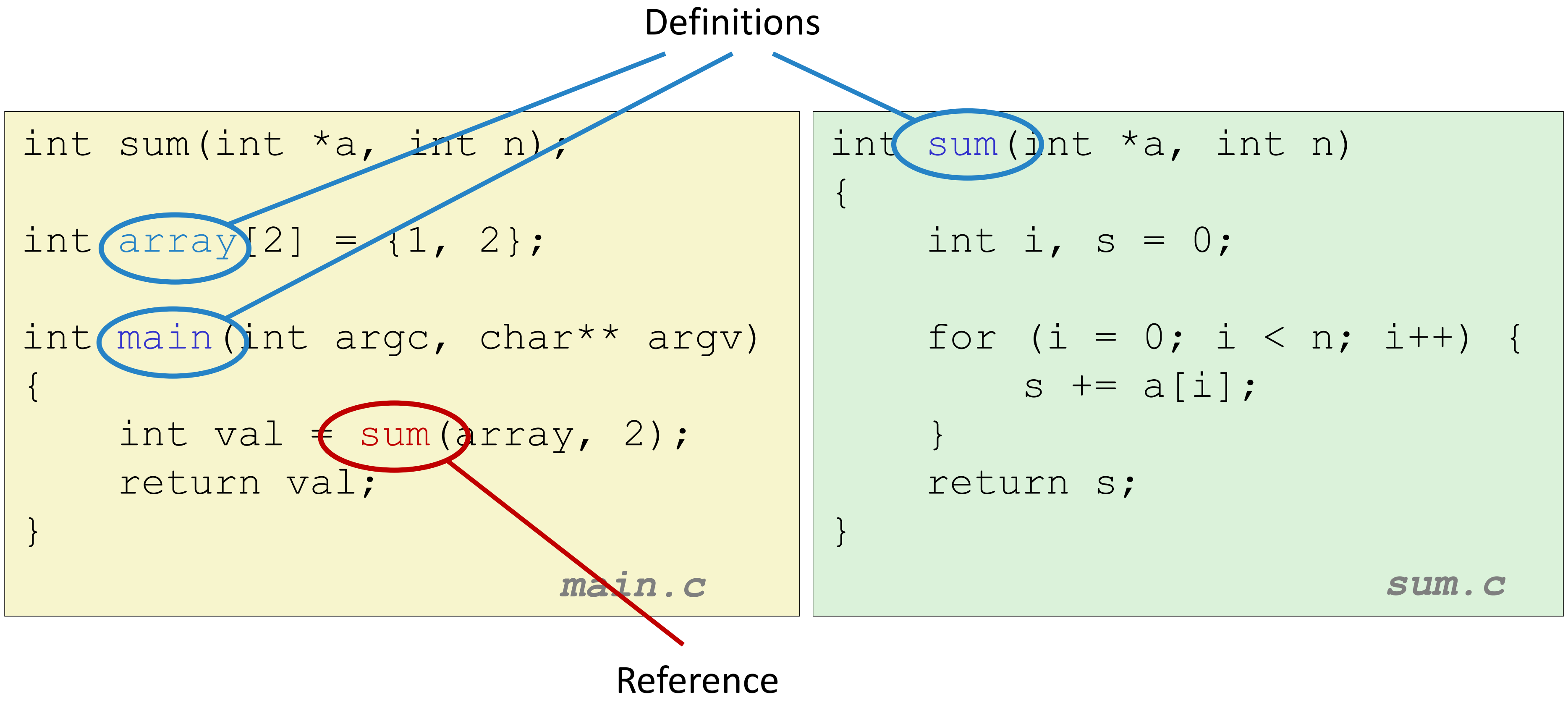
… THREE CASES

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 we will see one tricky way that it can be done, and even be useful!

SYMBOLS IN EXAMPLE C PROGRAM



Definitions

```
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

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

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

sum.c

Reference

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,

Segment header table

- ➢ Page size, virtual address memory segments + sizes.

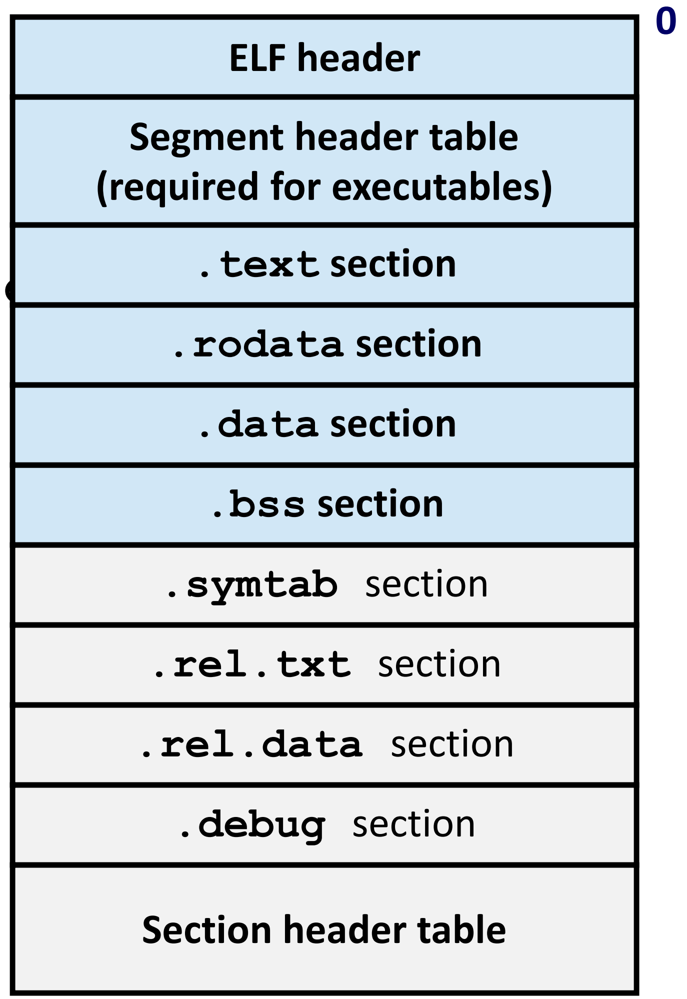
.text section (code)

.rodata section (read-only data, jump offsets, strings)

.data section (initialized global variables)

.bss section (name “bss” is lost in history)

- ➢ Global variables that weren’t initialized: zeros.
- ➢ Has section header but occupies no space



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

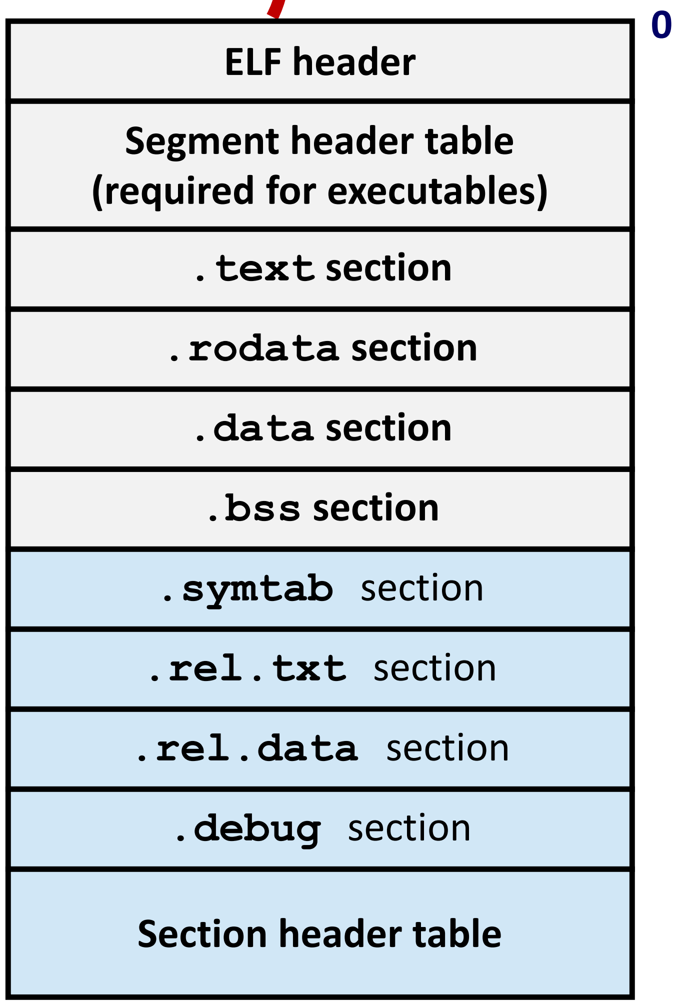
- Relocation info for .data section
- Addresses of pointer data that will need to be modified in the merged executable

.debug section

- Info for symbolic debugging (gcc -g)

Section header table

- Offsets and sizes of each section



0

ELF header

Segment header table
(required for executables)

.text section

.rodata section

.data section

.bss section

.symtab section

.rel.txt section

.rel.data section

.debug section

Section header table

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.

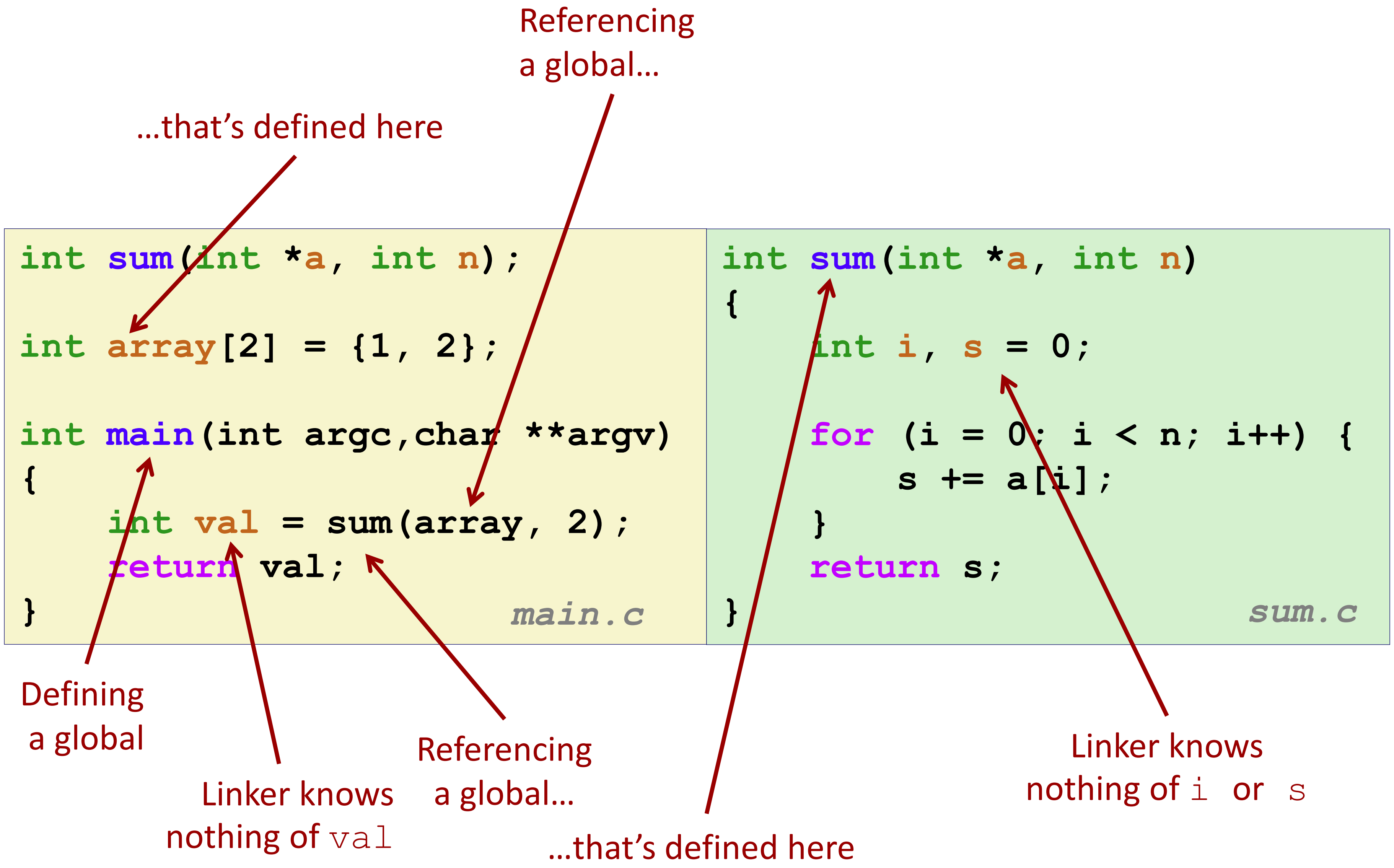
External symbols

- 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

EXAMPLE OF SYMBOL RESOLUTION



```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

Referencing a global…

…that's defined here

Defining a global

Linker knows nothing of `val`

Referencing a global…

…that's defined here

Linker knows nothing of `i` or `s`

SYMBOL IDENTIFICATION

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

symbols.**c**:

```
int incr = 1;
static 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;
}
```

static-local.c

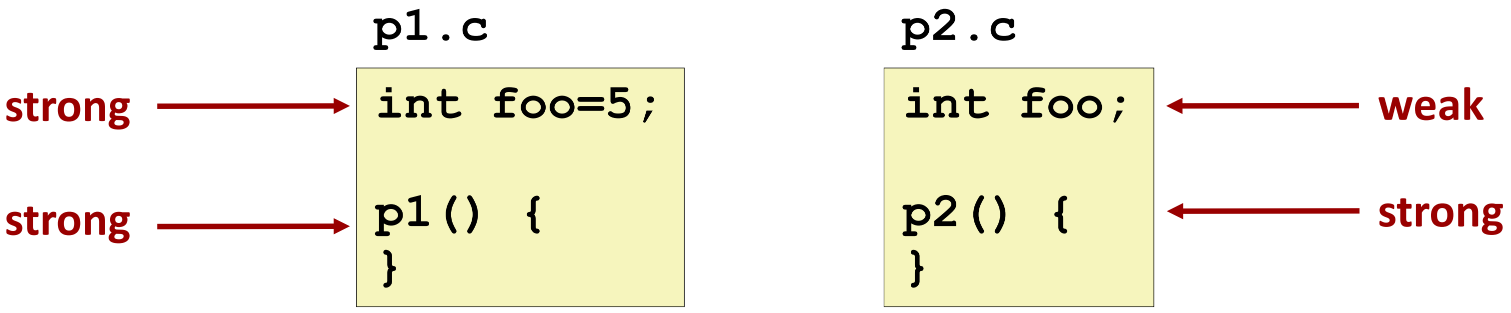
Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.1721` and `x.1724`.

HOW LINKER RESOLVES DUPLICATE SYMBOL DEFINITIONS

Program symbols are either strong or weak

- Strong: methods (code blocks) and initialized globals
- Weak: uninitialized globals (or with specifier extern)



p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

… but be aware that the “weak” case can cause real trouble!

LINKER WITH MULTIPLE WEAK DECLARATIONS

`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 might overwrite y!
		Nasty!

`int x=7;`
`p1() {}`
	`int x;`
	`p2() {}`
		References to x will refer to the same initialized variable.

Important: Linker does not do type checking. But C++ “namespaces” create a private naming scope.

GLOBAL TYPE MISMATCHES CAUSE BUGS

```
long int x;  /* Weak symbol */

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

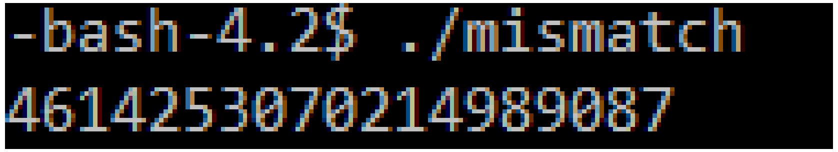
mismatch-main.c

```
/* Global strong symbol */
double x = 3.14;
```

mismatch-variable.c

Compiles without any errors or warnings, yet this is a bug!

What gets printed?



```
-bash-4.2$ ./mismatch
4614253070214989087
```

LINKING EXAMPLE

C++ won’t check to confirm that this array actually has n elements! The pointer (to **array[]**) that sum received doesn’t tell C++ anything about the underlying object type or size…

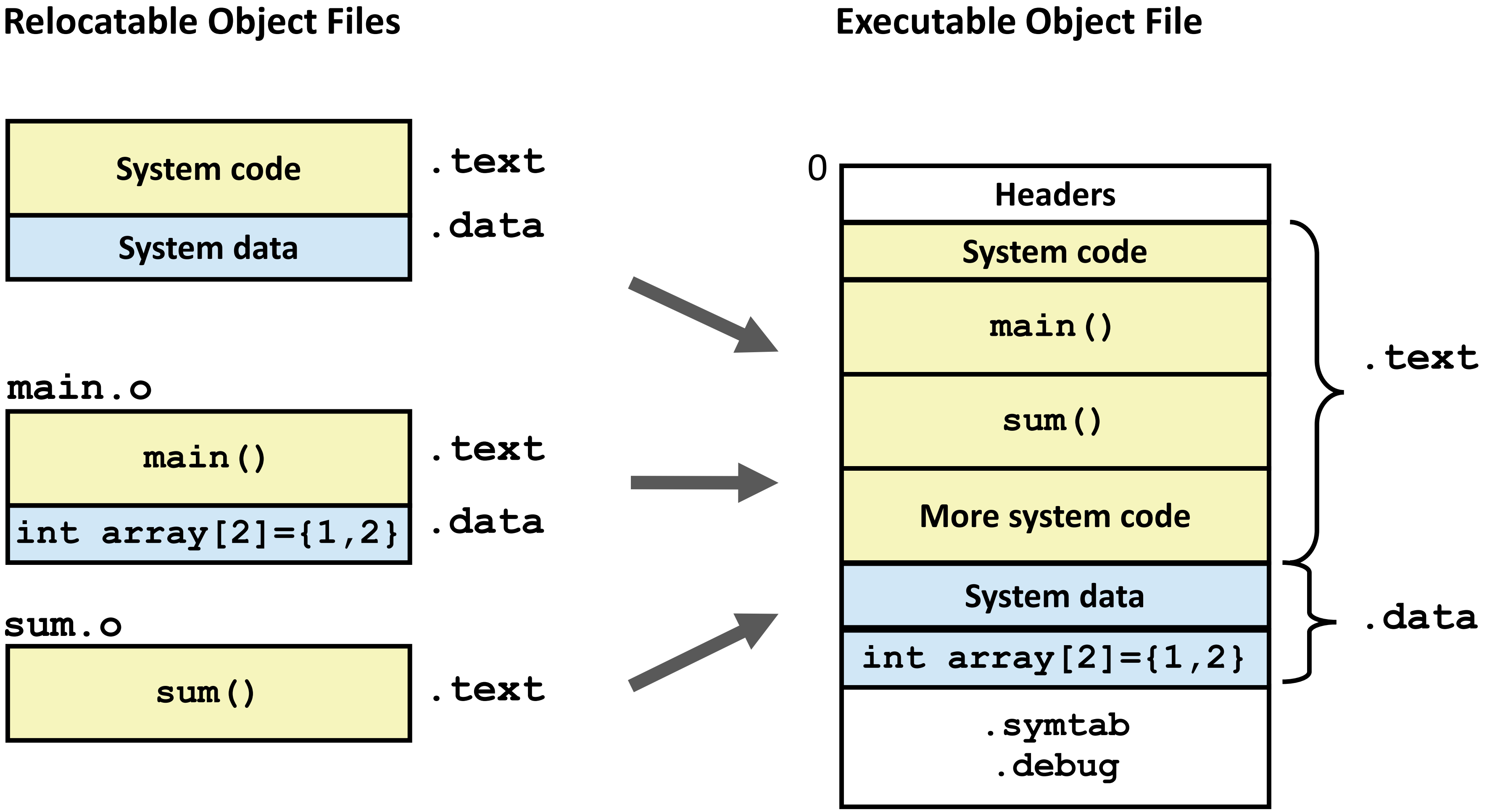
```
int sum(int *a, int n);

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

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

main.c

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

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

sum.c

STEP 2: RELOCATION



RELOCATION ENTRIES

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

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

main.c

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi      # %edi = &array
                        a: R_X86_64_32 array          # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4      # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

main.o

Source: **`objdump -r -d main.o`**

RELOCATED .TEXT SECTION

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08          sub    $0x8,%rsp
  4004d4:       be 02 00 00 00       mov    $0x2,%esi
  4004d9:       bf 18 10 60 00       mov    $0x601018,%edi   # %edi = &array
  4004de:       e8 05 00 00 00       callq  4004e8 <sum>     # sum()
  4004e3:       48 83 c4 08          add    $0x8,%rsp
  4004e7:       c3                   retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00             mov    $0x0,%eax
  4004ed:       ba 00 00 00 00             mov    $0x0,%edx
  4004f2:       eb 09                      jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                   movslq %edx,%rcx
  4004f7:       03 04 8f                   add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                   add    $0x1,%edx
  4004fd:       39 f2                      cmp    %esi,%edx
  4004ff:       7c f3                      jl     4004f4 <sum+0xc>
  400501:       f3 c3                      repz retq
```

`callq` instruction uses PC-relative addressing for sum():

0x4004e8 = 0x4004e3 + 0x5

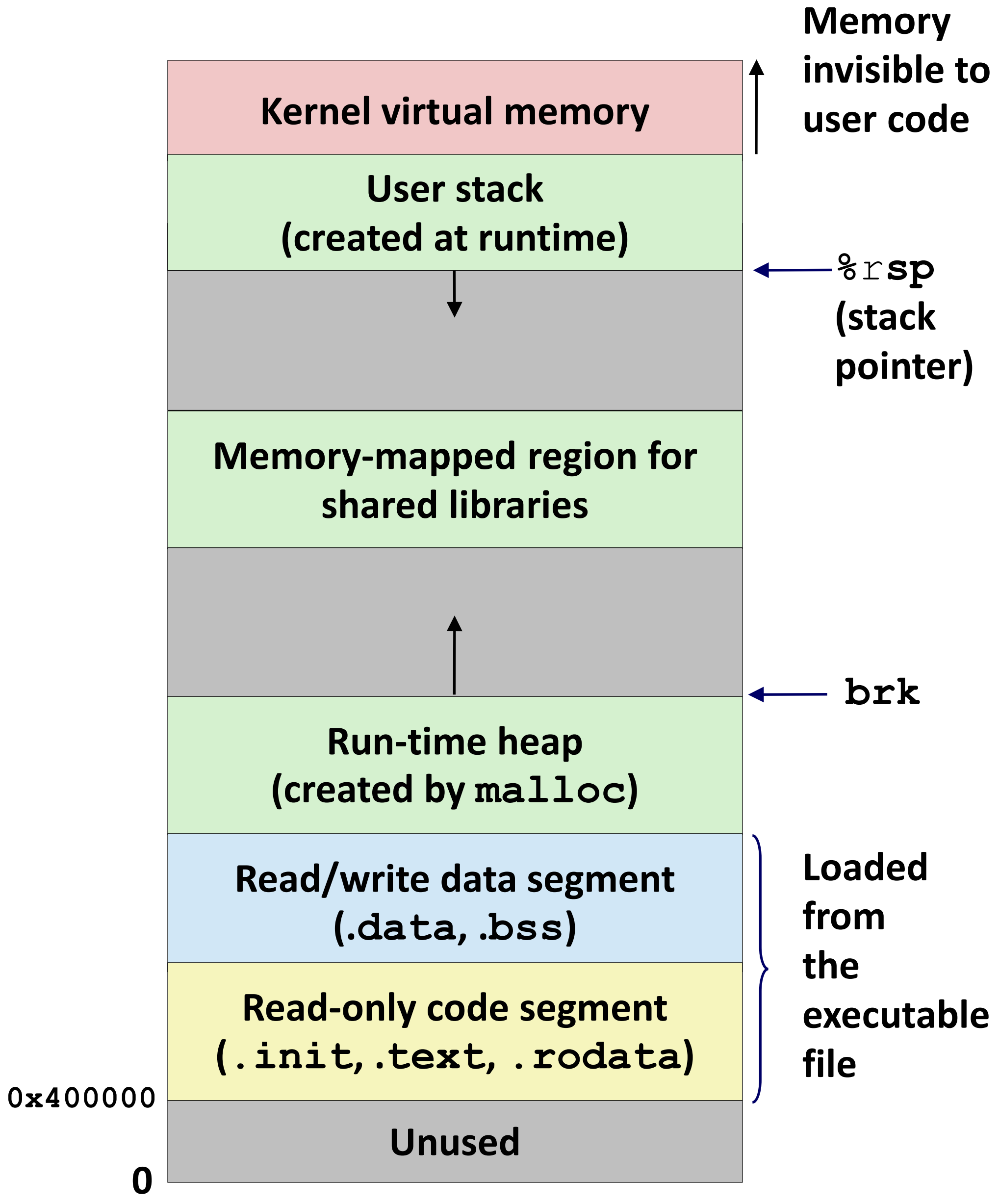
Source: objdump -d prog

LOADING EXECUTABLE OBJECT FILES

Executable Object File

0

- ELF header
- Program header table (required for executables)
- .init section
- .text section
- .rodata section
- .data section
- .bss section
- .symtab
- .debug
- .line
- .strtab
- Section header table (required for relocatables)



Kernel virtual memory

Memory invisible to user code

User stack (created at runtime)

`%rsp` (stack pointer)

Memory-mapped region for shared libraries

`brk`

Run-time heap (created by `malloc`)

Read/write data segment (`.data`, `.bss`)

Read-only code segment (`.init`, `.text`, `.rodata`)

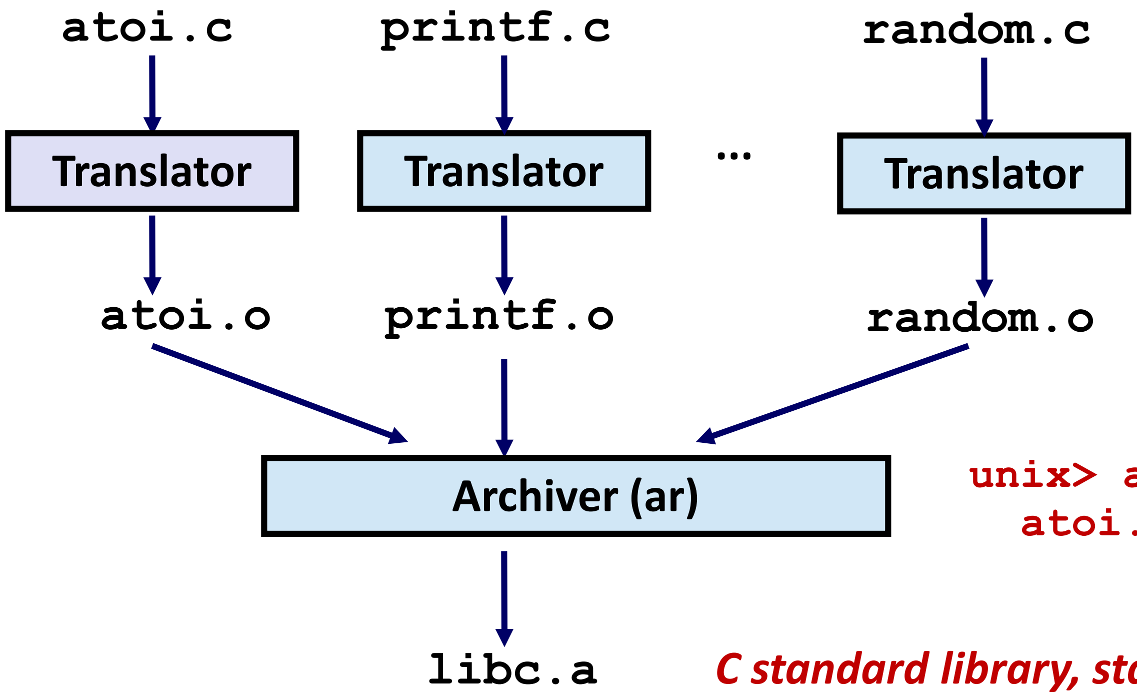
Loaded from the executable file

`0x400000`

Unused

0

STATIC LIBRARIES



atoi.c printf.c … random.c

Translator Translator Translator

atoi.o printf.o random.o

Archiver (ar)

libc.a

```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

C standard library, static version

- 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, …)

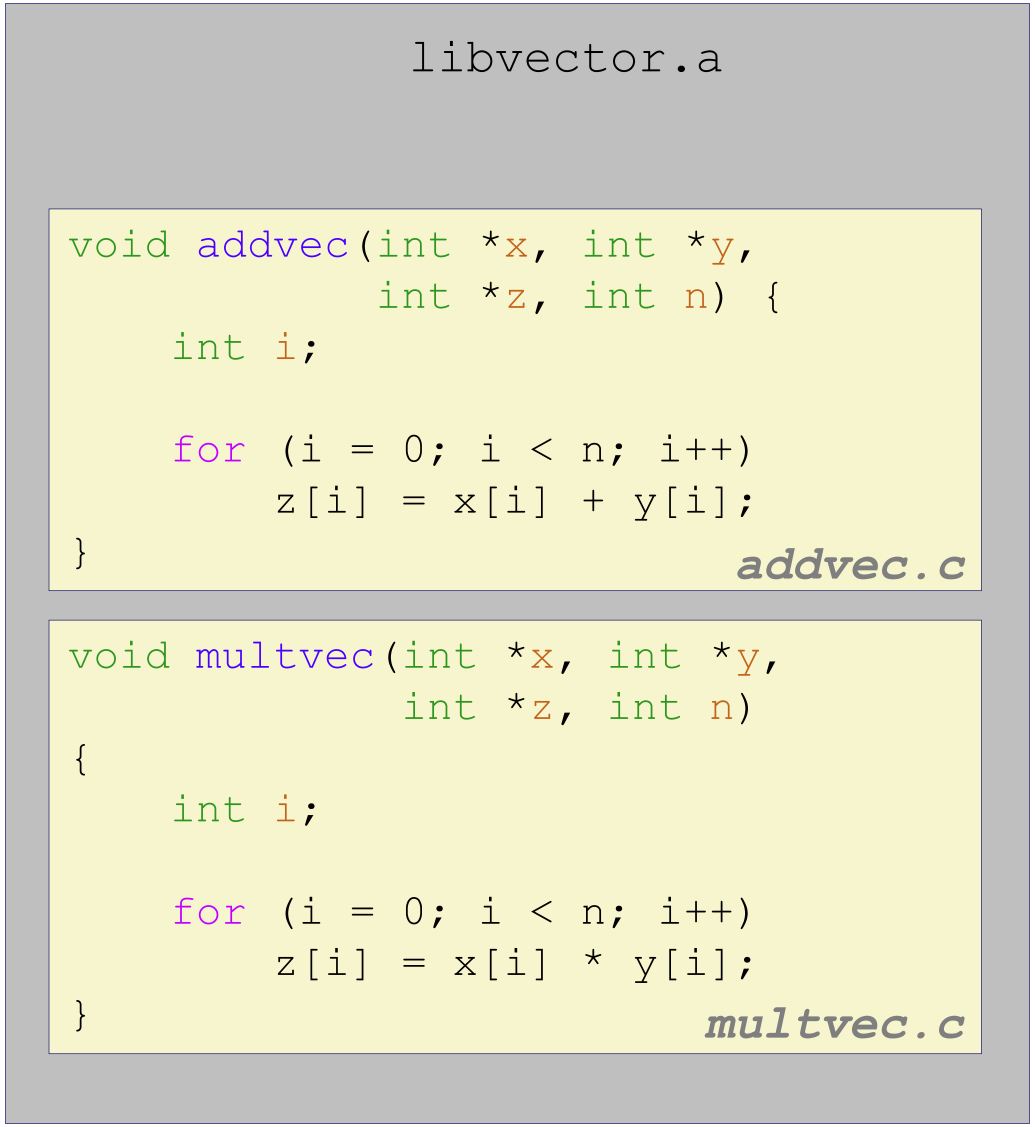
```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% 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

```
#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

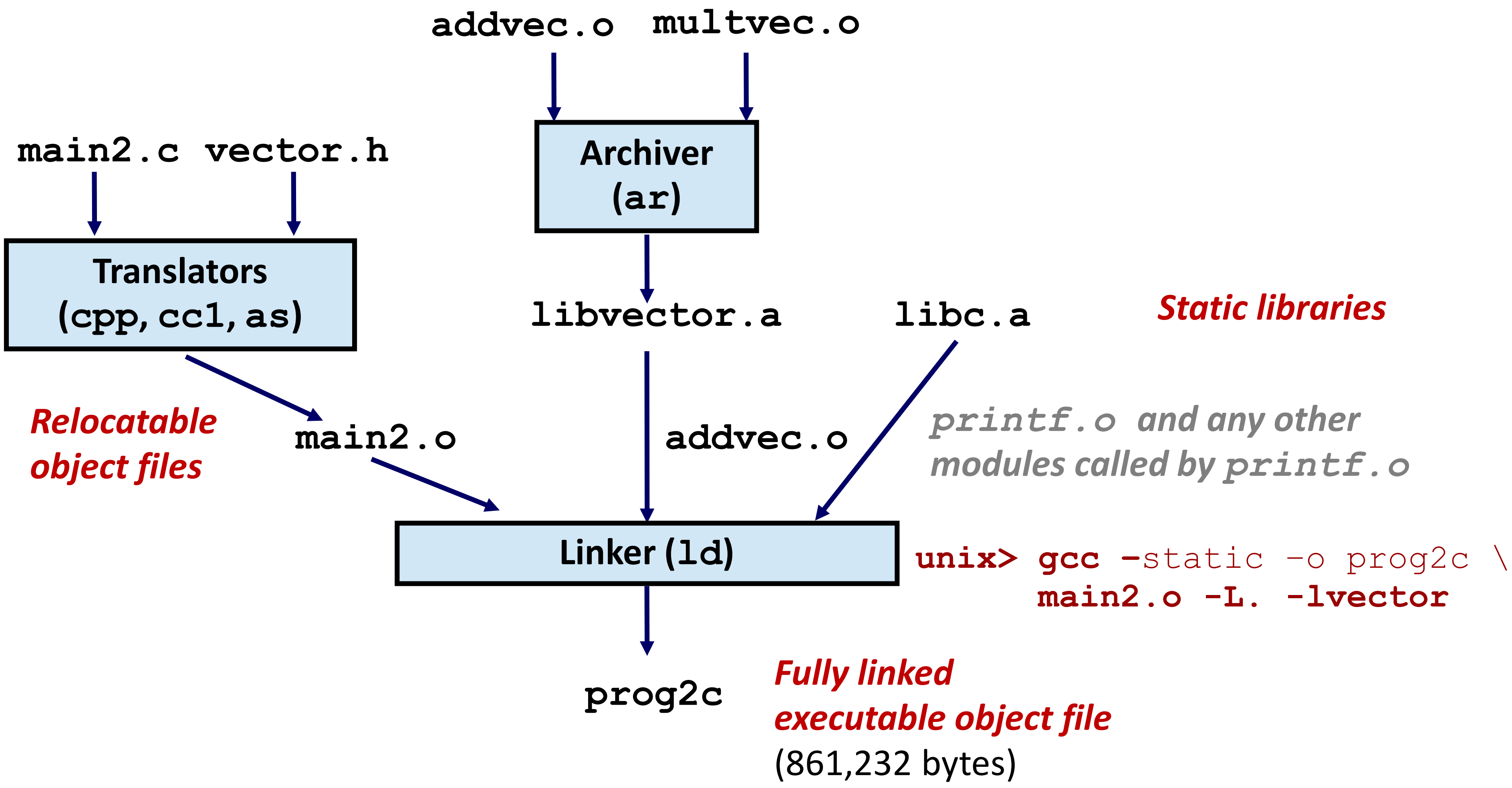
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    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



addvec.o multvec.o

Archiver (ar)

main2.c vector.h

Translators (cpp, cc1, as)

libvector.a libc.a *Static libraries*

Relocatable object files

main2.o

addvec.o

printf.o and any other modules called by printf.o

Linker (ld)

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

prog2c

Fully linked executable object file
(861,232 bytes)

“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 -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

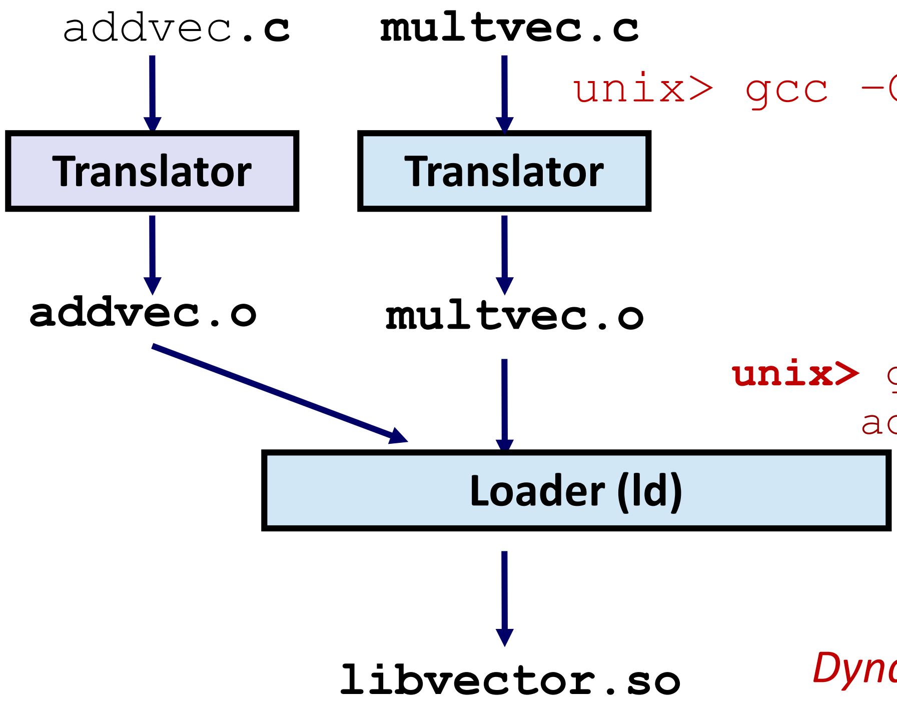
SHARED LIBRARIES

Shared libraries save space and resolve this issue.

Term refers to:

- Object files that contain code and data.
- Saved in a special directly (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

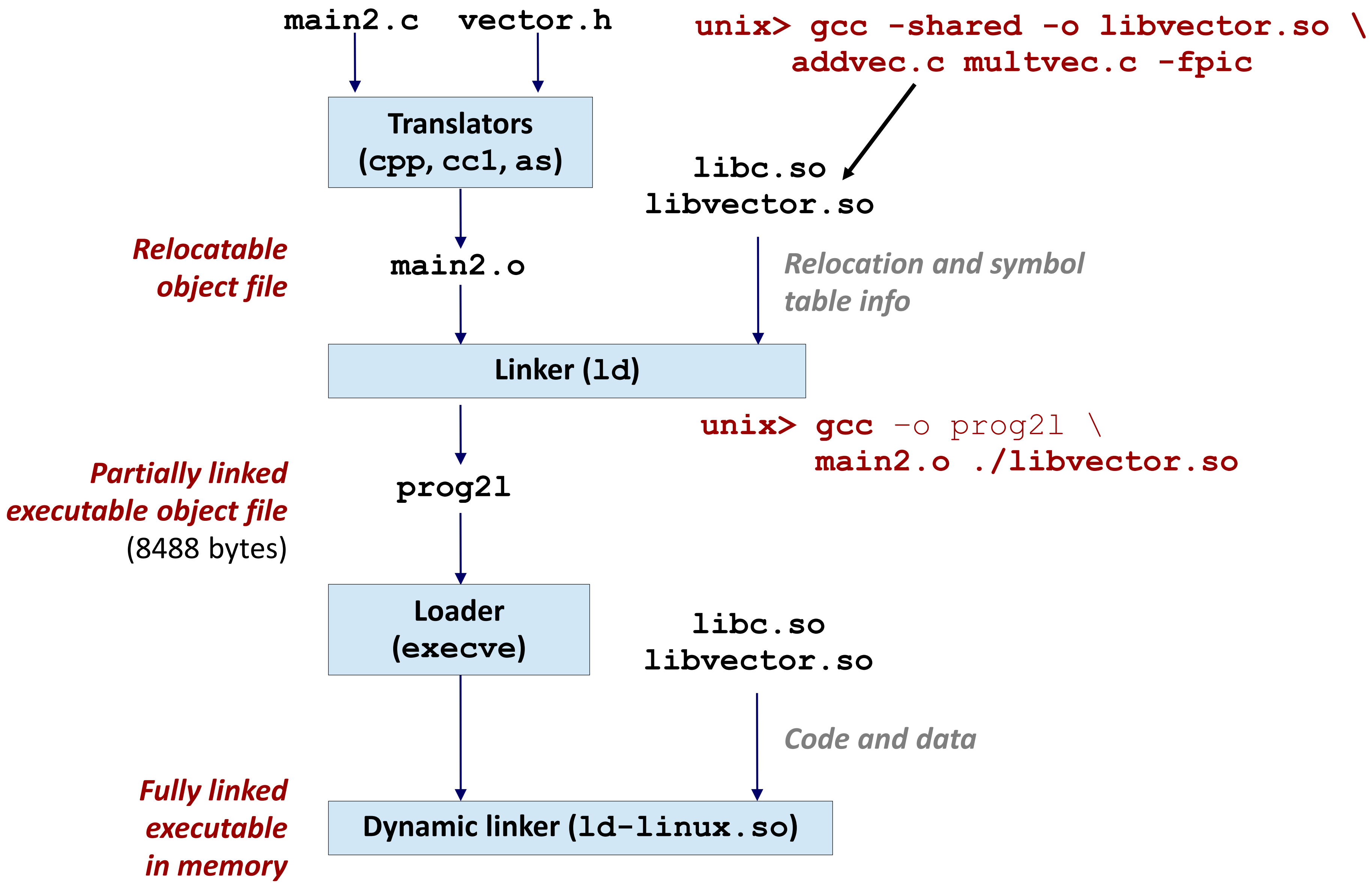


```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

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

Dynamic vector library

DYNAMIC LINKING AT LOAD-TIME



main2.c

vector.h

Translators (cpp, cc1, as)

```
unix> gcc -shared -o libvector.so \
        addvec.c multvec.c -fpic
```

libc.so
libvector.so

Relocatable object file

main2.o

Relocation and symbol table info

Linker (ld)

```
unix> gcc -o prog2l \
      main2.o ./libvector.so
```

Partially linked executable object file
(8488 bytes)

prog2l

Loader (execve)

libc.so
libvector.so

Code and data

Fully linked executable in memory

Dynamic linker (ld-linux.so)

FOR DYNAMIC LINKING, RELOCATION OCCURS AT RUNTIME

If a program uses a library, the operating system maps it into memory. The single copy can then be shared

Then a “dynamic linking” module runs to connect the executable to the mapped library segment.

- It may have a different base address in each address space, creating a need for *dynamic relocation.*
- We also create a copy of the data segments of the library for each process using it, so that any changes are private.

DYNAMIC LINKING AT RUN-TIME

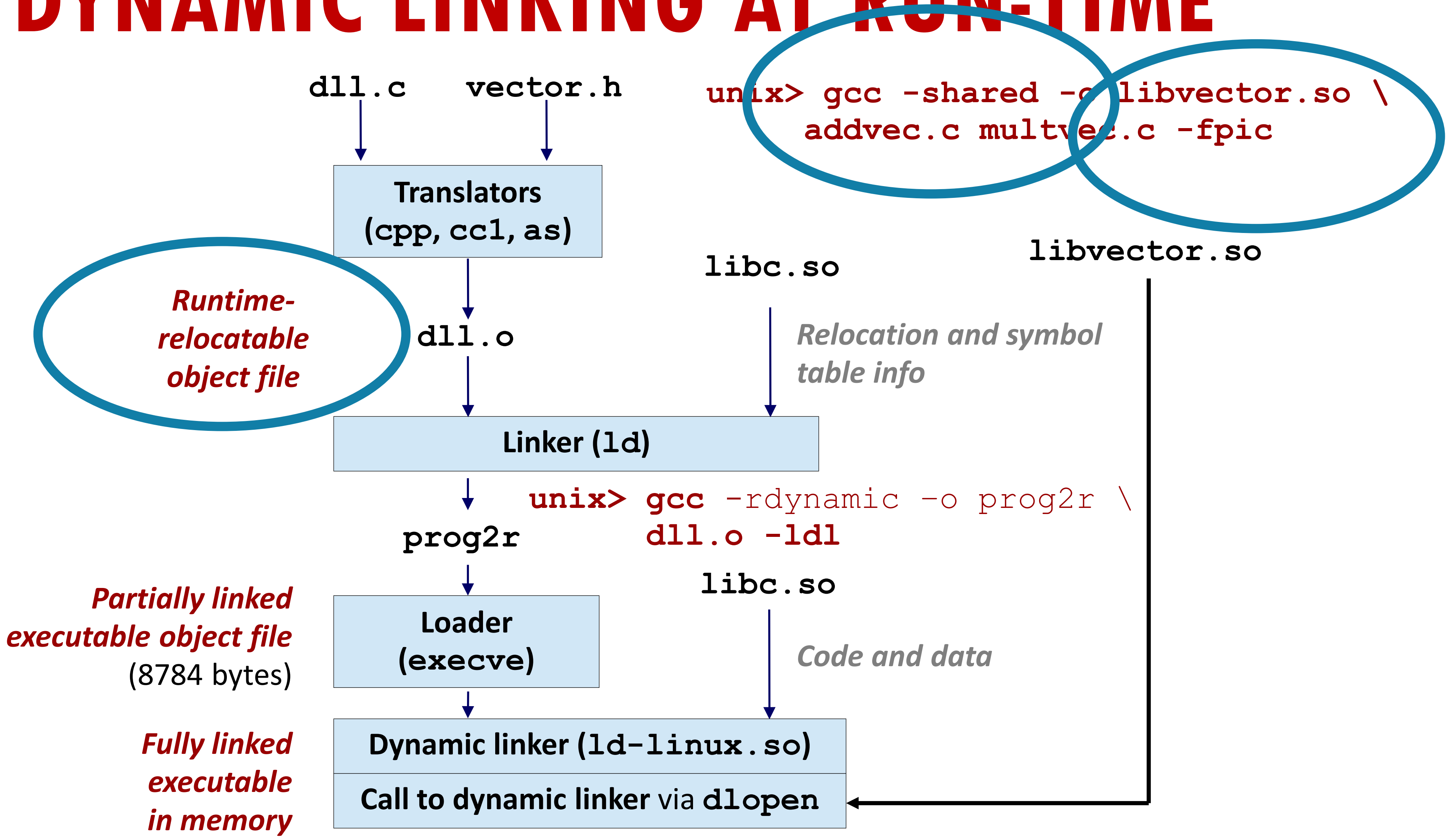
```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

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'D)

```c
    ...

    /* 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



```
unix> gcc -shared -o libvector.so \
  addvec.c multvec.c -fpic
```

```
unix> gcc -rdynamic -o prog2r \
      dll.o -ldl
```

dll.c
vector.h
Translators (cpp, cc1, as)
Runtime-relocatable object file
dll.o
libc.so
libvector.so
Relocation and symbol table info
Linker (ld)
prog2r
Partially linked executable object file (8784 bytes)
libc.so
Code and data
Loader (execve)
Fully linked executable in memory
Dynamic linker (ld-linux.so)
Call to dynamic linker via dlopen

GCC OPTIONS USED HERE

1) –shared, -fpic: To create position independent code (next slide)

2) –o something.so: To output result as a DLL

3) –rdynamic: Includes dynamic symbol names for gprof, gdb

4) –ldr: “dr” is the directory to look for the .so file in

DYNAMIC LOADING REQUIRES THAT THE SHARED LIBRARY BE RELOCATABLE, BUT MORE…

With mapped files (Linux mmap API), the segment can be a different base address in each process.

So… not only does each process see the DLL at a different location in memory, the DLL sees *itself* there too!

And in fact each also has its own data segment

SOLUTION INVOLVES TWO ASPECTS

We compile the library with –shared –fPIC. This tells the compiler to generate "register offset" addressing

Then, at runtime, whenever we call into the shared library, we need to put the code segment base address in a specific register (save the old value to the stack!), and the data segment base into a second register (" " "). Restore the original values when the method returns.

With –fPIC, all jumps and data accesses in the DLL are "relativized" as offsets with respect to these registers.

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

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

GETTING VERY FANCY: LIBRARY INTERPOSITIONING (FOR SERIOUS HACKERS!)

Documented in Section 7.13 of book

Library interpositioning: powerful linking technique that allows programmers to intercept calls to arbitrary functions

Interpositioning can occur at:

- Compile time: When the source code is compiled
- Link time: When the relocatable object files are statically linked to form an executable object file
- Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

1-2-3 RECIPE FOR INTERPOSITIONING

Given an executable that obtains **something** from a library.

Create a .o file that defines **something,** using the same API the executable expected. Relink the executable against your .o file.

Now your implementation of **something** will be called

1-2-3 RECIPE FOR INTERPOSITIONING

… but what if you wanted to call the standard **something** from inside your replacement?

If it were to call **something**, that would just be a recursive call.

… So, have it call _**something.** This will be undefined… claim that it is in a library

1-2-3 RECIPE FOR INTERPOSITIONING

So now we have the original executable, and it calls your version of **something**, which calls _**something.**

Create a new DLL library that defines _**something.** It calls the original **something, from the original DLL.**

Now we have "wrapped" **something**!

… SHORTCUT

There are also linker arguments you can use to just tell the linker you wish to wrap some method.

Eliminates the need to create the extra helper DLL.

Time permitting, I'll show you an example that wraps malloc

SOME INTERPOSITIONING APPLICATIONS

Security

- Confinement (sandboxing)
- Behind the scenes encryption

Debugging

- In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
- Code in the SPDY networking stack was writing to the wrong location
- Solved by intercepting calls to Posix write functions (write, writev, pwrite)

- Source: Facebook engineering blog post at:
- https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/

SOME INTERPOSITIONING APPLICATIONS

Monitoring and Profiling

- Count number of calls to functions
- Characterize call sites and arguments to functions
- Malloc tracing
  - Detecting memory leaks
  - Generating address traces

Changing a local resource into one accessed over a network

EXAMPLE PROGRAM

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

int main(int argc,
         char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    void *p =
          malloc(atoi(argv[i]));
    free(p);
  }
  return(0);
}
```

int.c

Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.

Three solutions: interpose on the library `malloc` and `free` functions at compile time, link time, and load/run time.

We won't cover this example if we are short on time; it is not required and you won't see questions about these slides on a quiz

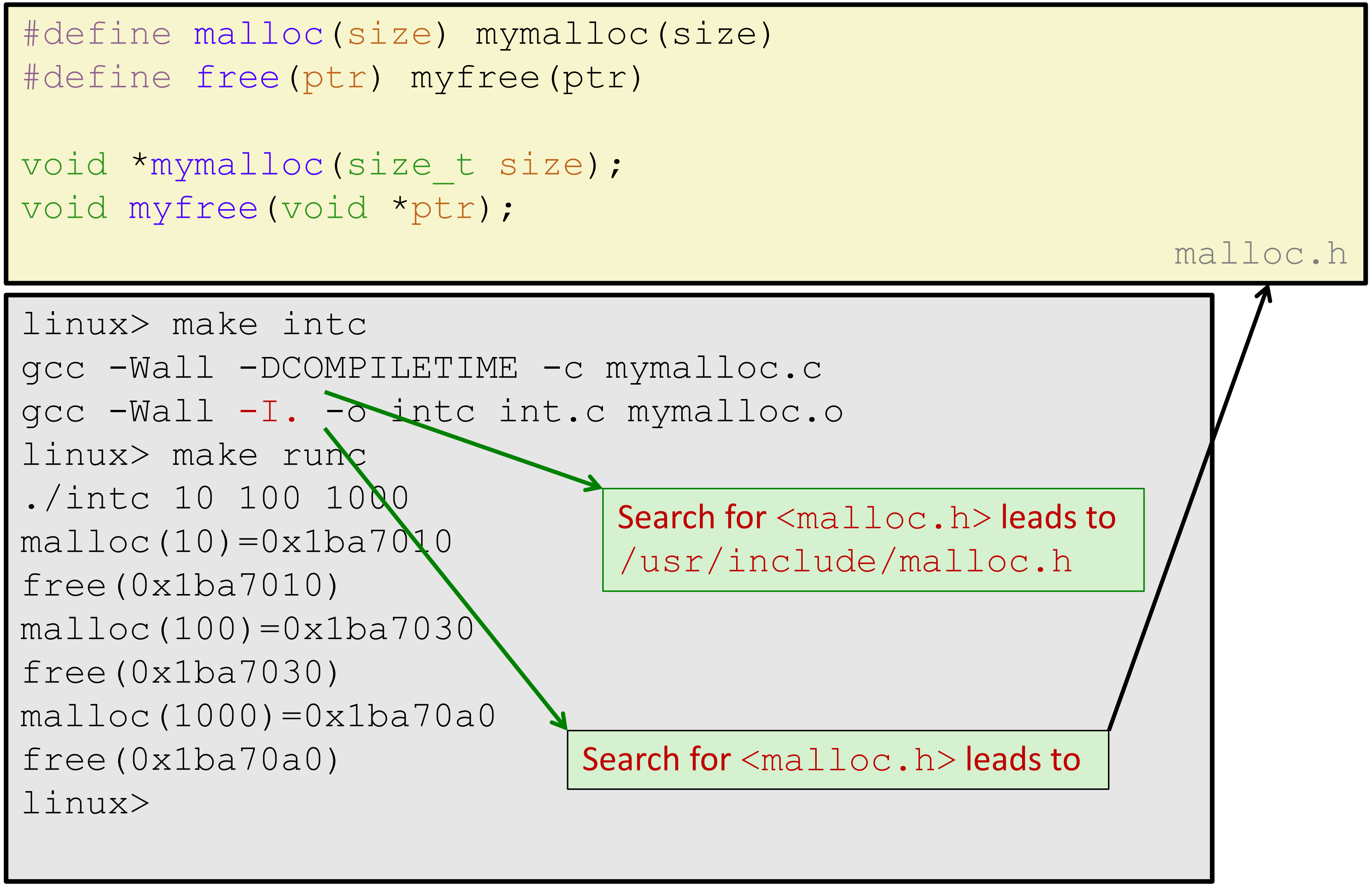
COMPILE-TIME INTERPOSITIONING

```c
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Time permitting

COMPILE-TIME INTERPOSITIONING



```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc 10 100 1000
malloc(10)=0x1ba7010
free(0x1ba7010)
malloc(100)=0x1ba7030
free(0x1ba7030)
malloc(1000)=0x1ba70a0
free(0x1ba70a0)
linux>
```

Search for <malloc.h> leads to /usr/include/malloc.h

Search for <malloc.h> leads to

Time permitting

LINK-TIME INTERPOSITIONING

```
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Time permitting

LINK-TIME INTERPOSITIONING

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl \
    int.o mymalloc.o
linux> make runl
./intl 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
. . .
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

The "`-Wl`" flag passes argument to linker, replacing each comma with a space.

The "`--wrap,malloc`" `arg` instructs linker to resolve references in a special way:

- Refs to `malloc` should be resolved as `__wrap_malloc`
- Refs to `__real_malloc` should be resolved as `malloc`

Time permitting

LOAD/RUN-TIME INTERPOSITIONING

Observe that we DON'T have `#include <malloc.h>`

```c
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Time permitting

LOAD/RUN-TIME INTERPOSITIONING

```c
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Time permitting

LOAD/RUN-TIME INTERPOSITIONING

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr 10 100 1000)
malloc(10) = 0x91a010
free(0x91a010)
. . .
linux>
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

`The LD_PRELOAD` **environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `mymalloc.so` first.**

Type into (some) shells as:

```
env LD_PRELOAD=./mymalloc.so ./intr 10 100 1000)
```

Time permitting

INTERPOSITIONING RECAP

Compile Time

- Apparent calls to **malloc**/**free** get macro-expanded into calls to **mymalloc**/**myfree**
- Simple approach. Must have access to source & recompile

Link Time

- Use linker trick to have special name resolutions
  - **malloc → __wrap_malloc**
  - **__real_malloc → malloc**

Load/Run Time

- Implement custom version of **malloc**/**free** that use dynamic linking to load library **malloc**/**free** under different names
- Can use with ANY dynamically linked binary

```
env LD_PRELOAD=./mymalloc.so gcc -c int.c)
```

Time permitting

LINKING SUMMARY

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