Linking

CSCI 2021: Machine Architecture and Organization

Antonia Zhai

Department Computer Science and Engineering

University of Minnesota

http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

University of Minnesota

Linker Puzzles

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

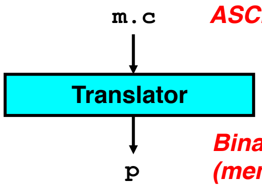
```
int x;        int x;
p1() {}       p2() {}
```

```
int x;        double x;
int y;        p2() {}
p1() {}
```

```
int x=7;      double x;
int y=5;      p2() {}
p1() {}
```

```
int x=7;      int x;
p1() {}       p2() {}
```

A Simplistic Program Translation Scheme



`m.c`

Translator

`p`

ASCII source file

Binary executable object file (memory image on disk)

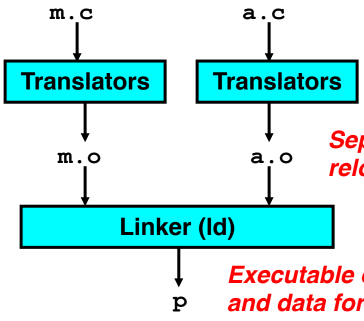
Problems:

- Efficiency: small change requires complete recompilation
- Modularity: hard to share common functions (e.g. `printf`)

Solution:

- *Static linker (or linker)*

A Better Scheme Using a Linker



`m.c` `a.c`

Translators Translators

`m.o` `a.o`

Linker (ld)

`p`

Separately compiled relocatable object files

Executable object file (contains code and data for all functions defined in `m.c` and `a.c`)

Translating the Example Program

Compiler driver coordinates all steps in the translation and linking process.

- Typically included with each compilation system (e.g., `gcc`)
- Invokes preprocessor (`cpp`), compiler (`cc1`), assembler (`as`), and linker (`ld`).
- Passes command line arguments to appropriate phases

Example: create executable `p` from `m.c` and `a.c`:

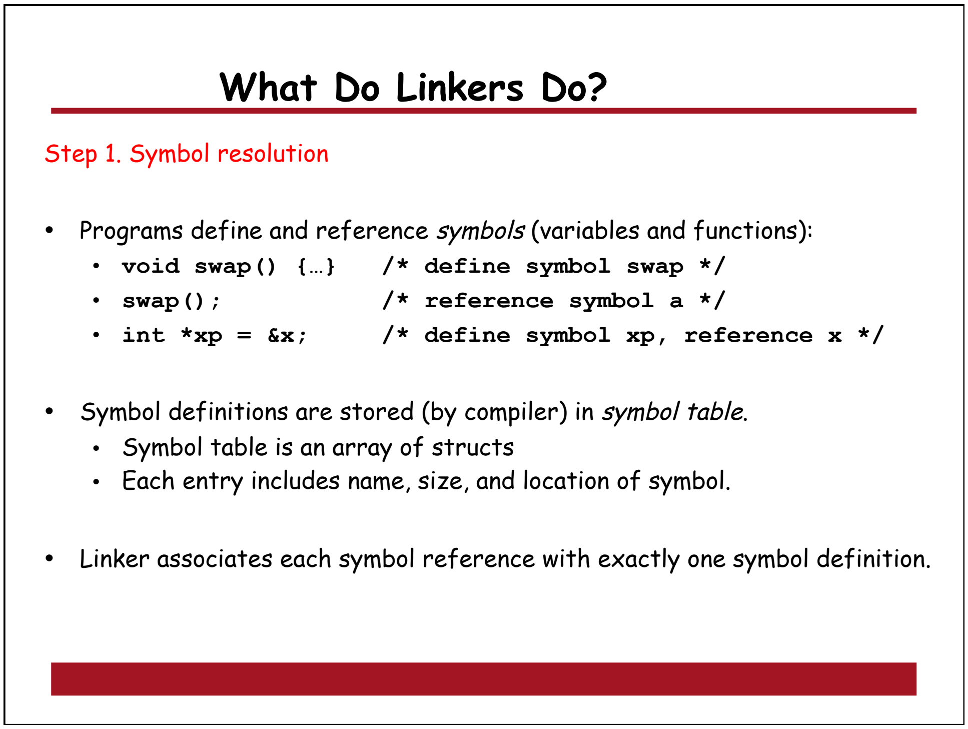
Why Linkers?

Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

Efficiency

- Time:
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.



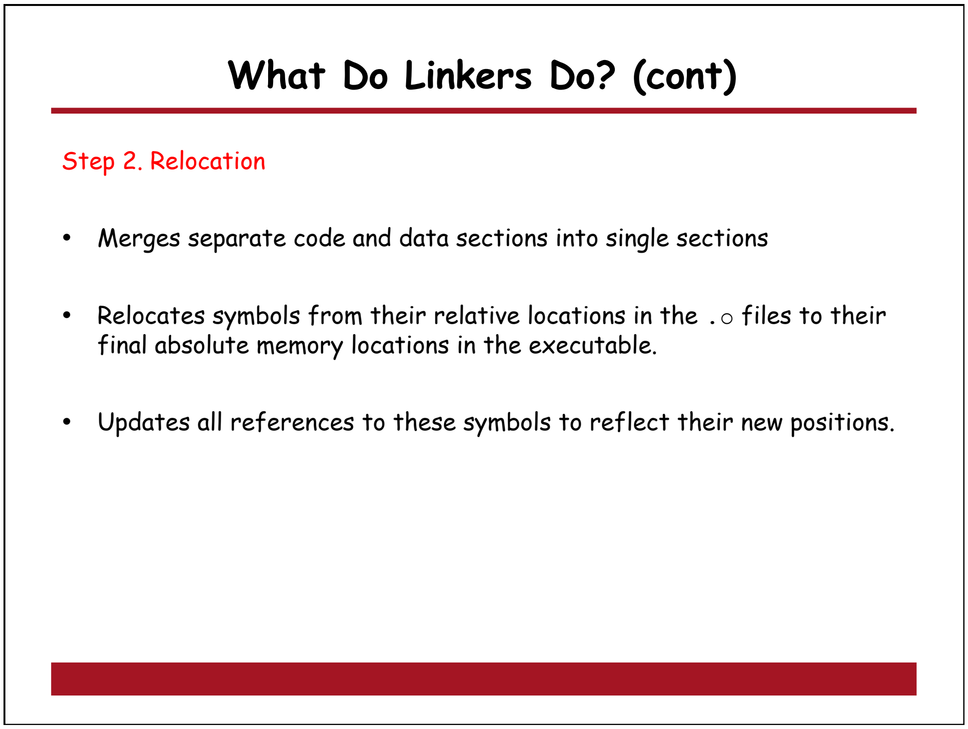
What Do Linkers Do?

Step 1. Symbol resolution

- Programs define and reference *symbols* (variables and functions):

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

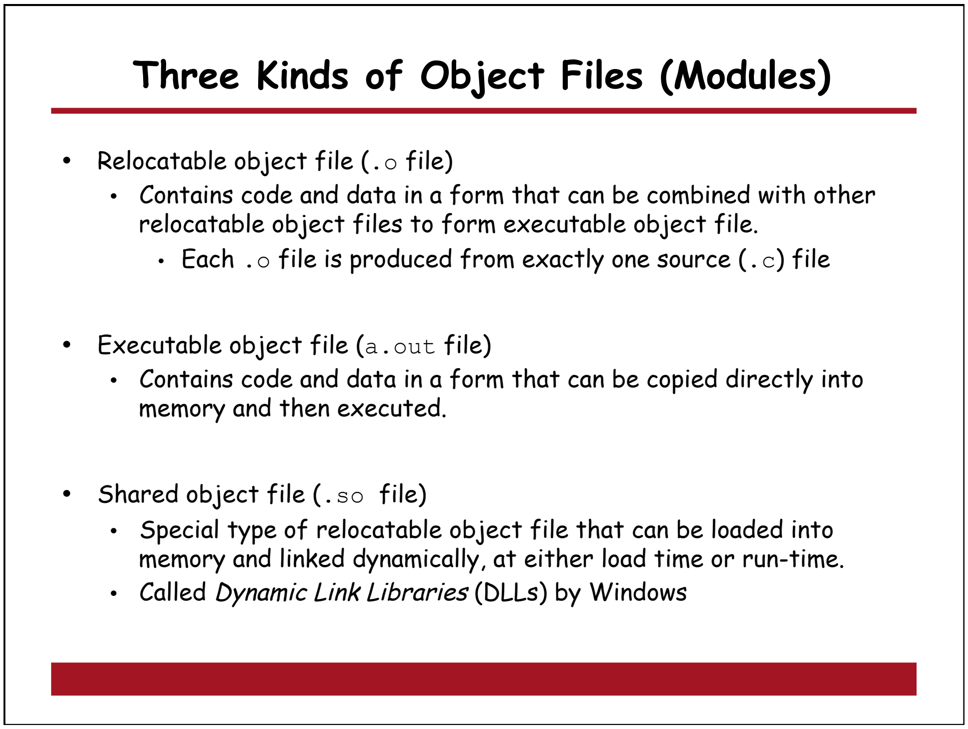
- Symbol definitions are stored (by compiler) in *symbol table*.
  - Symbol table is an array of structs
  - Each entry includes name, size, and location of symbol.
- Linker associates each symbol reference with exactly one symbol definition.



What Do Linkers Do? (cont)

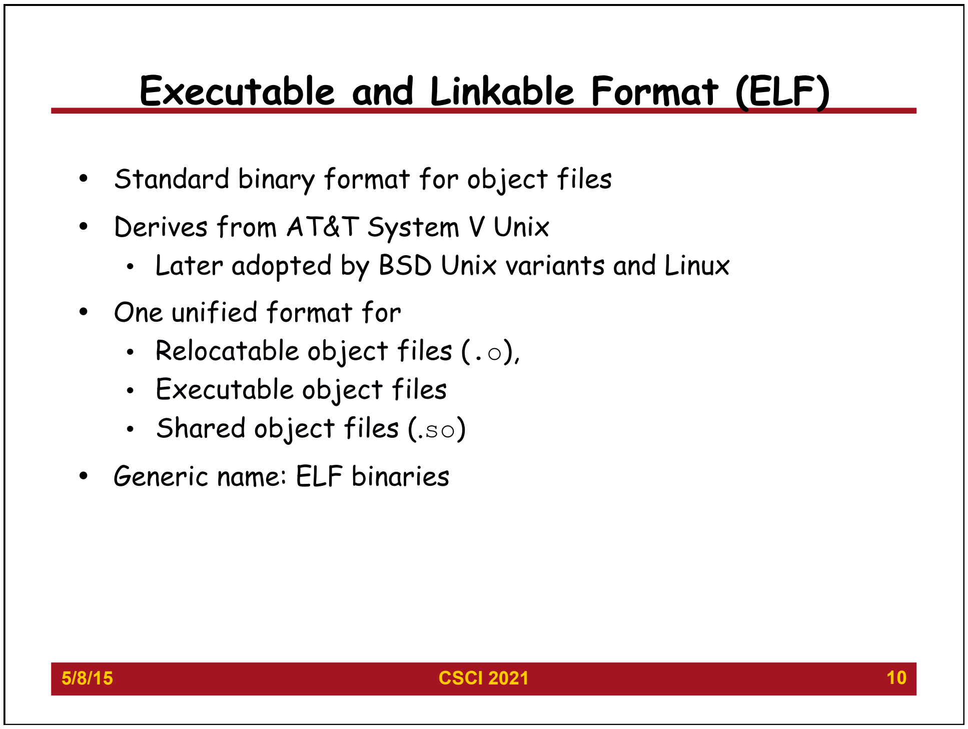
Step 2. Relocation

- 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.



Three Kinds of Object Files (Modules)

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

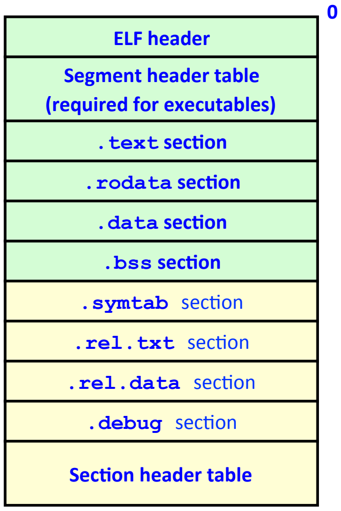


Executable and Linkable Format (ELF)

- Standard binary format for object files
- Derives from AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Relocatable object files (`.o`),
  - Executable object files
  - Shared object files (`.so`)
- Generic name: ELF binaries

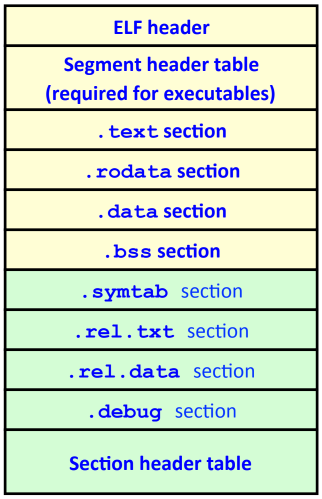
ELF Object File Format

- 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



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



Example C Program

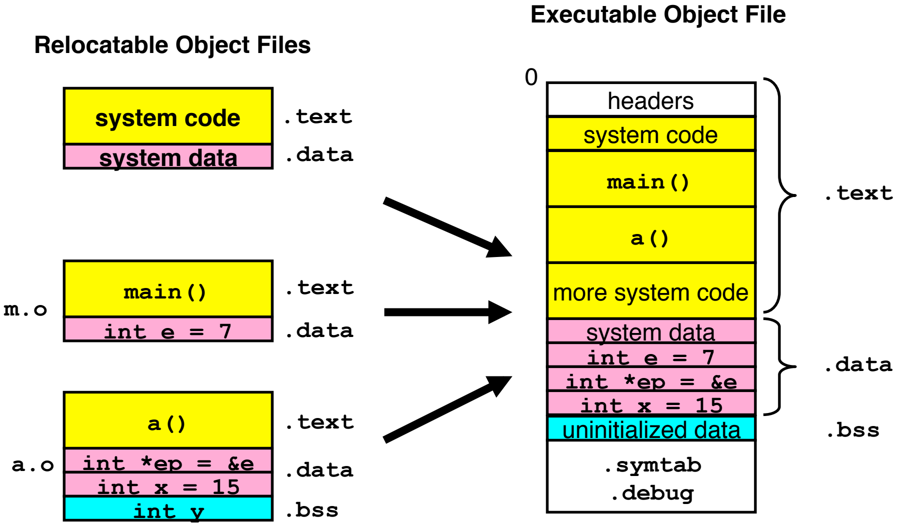
m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Merging Relocatable Object Files into an Executable Object File



Relocatable Object Files

system code .text
system data .data

m.o
main() .text
int e = 7 .data

a.o
a() .text
int *ep = &e .data
int x = 15
int y .bss

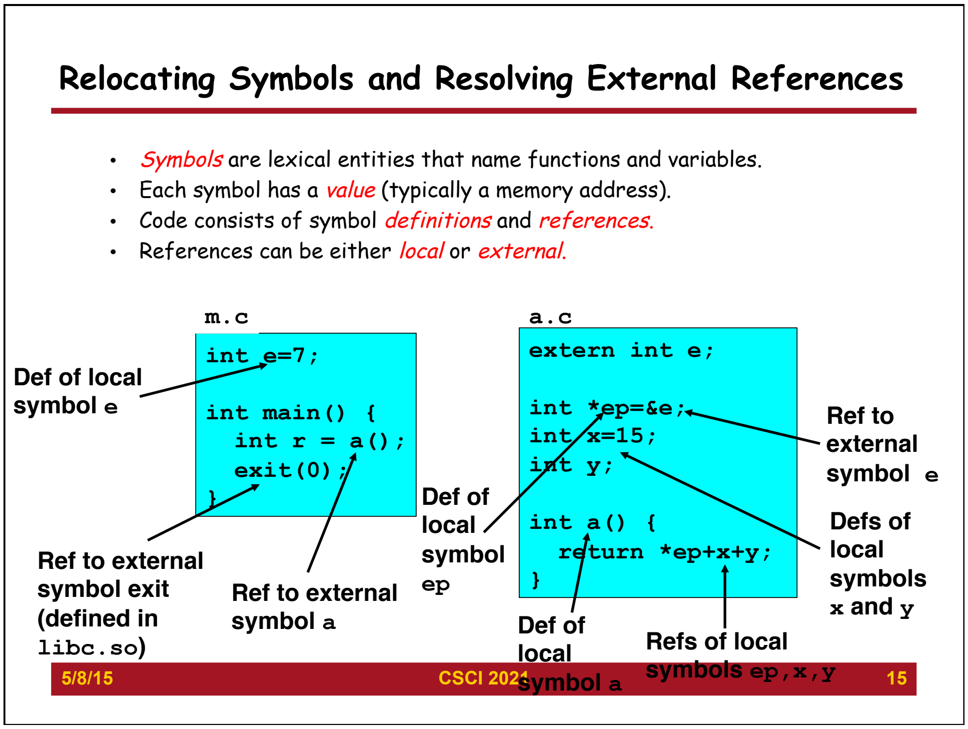
Executable Object File

0
headers
system code
main()
a()
more system code
.text

system data
int e = 7
int *ep = &e
int x = 15
.data

uninitialized data .bss

.symtab
.debug



Relocating Symbols and Resolving External References

- *Symbols* are lexical entities that name functions and variables.
- Each symbol has a *value* (typically a memory address).
- Code consists of symbol *definitions* and *references*.
- References can be either *local* or *external*.

m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Def of local symbol `e`

Ref to external symbol exit (defined in `libc.so`)

Ref to external symbol `a`

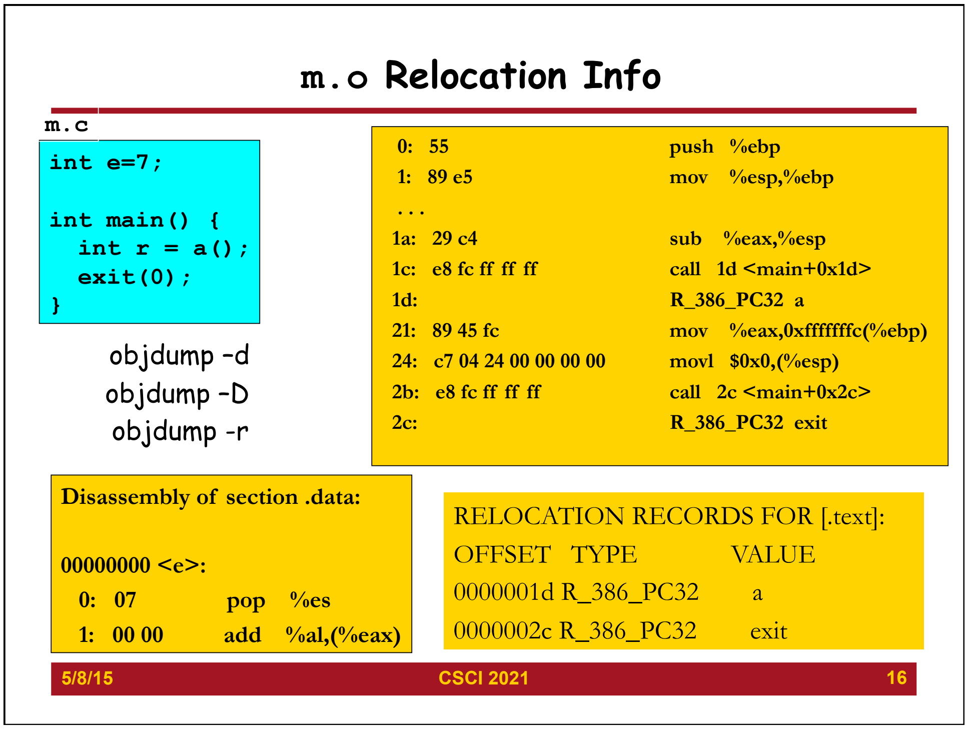
Def of local symbol `ep`

Ref to external symbol `e`

Defs of local symbols `x` and `y`

Def of local symbol `a`

Refs of local symbols `ep,x,y`



m.o Relocation Info

m.c

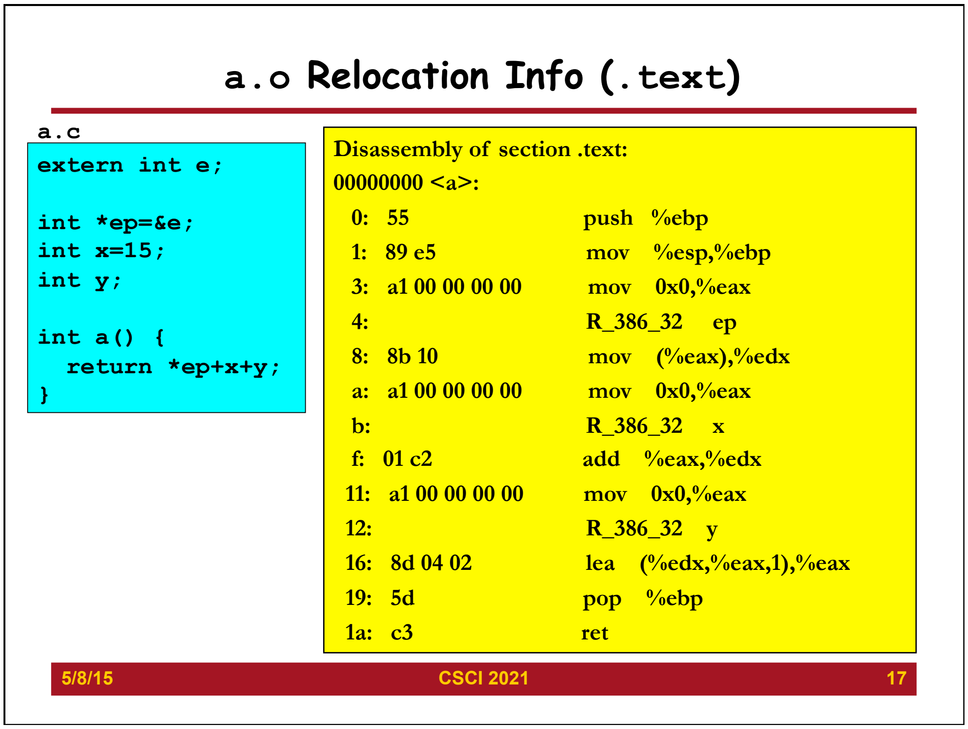
```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

objdump -d
objdump -D
objdump -r

```
 0:  55                      push  %ebp
 1:  89 e5                   mov   %esp,%ebp
 . . .
1a:  29 c4                   sub   %eax,%esp
1c:  e8 fc ff ff ff          call  1d <main+0x1d>
1d:                          R_386_PC32  a
21:  89 45 fc                mov   %eax,0xfffffffc(%ebp)
24:  c7 04 24 00 00 00 00    movl  $0x0,(%esp)
2b:  e8 fc ff ff ff          call  2c <main+0x2c>
2c:                          R_386_PC32  exit
```

```
Disassembly of section .data:

00000000 <e>:
  0:  07          pop   %es
  1:  00 00       add   %al,(%eax)
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE          VALUE
0000001d R_386_PC32     a
0000002c R_386_PC32     exit
```

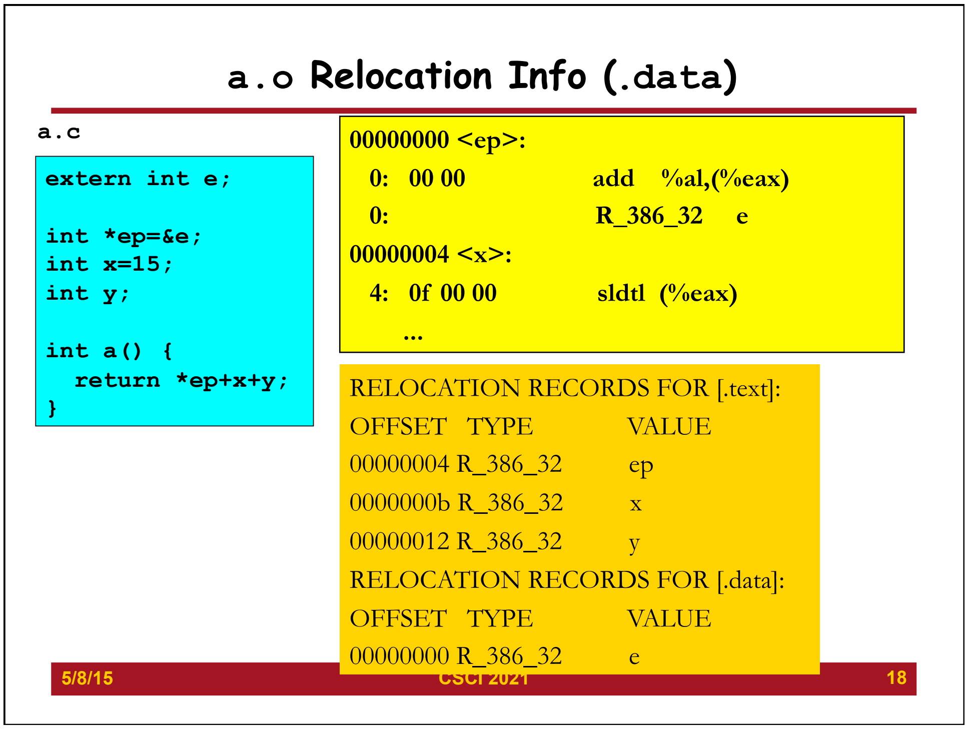


a.o Relocation Info (.text)

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .text:
00000000 <a>:
  0:  55               push  %ebp
  1:  89 e5            mov   %esp,%ebp
  3:  a1 00 00 00 00   mov   0x0,%eax
  4:                   R_386_32   ep
  8:  8b 10            mov   (%eax),%edx
  a:  a1 00 00 00 00   mov   0x0,%eax
  b:                   R_386_32   x
  f:  01 c2            add   %eax,%edx
 11:  a1 00 00 00 00   mov   0x0,%eax
 12:                   R_386_32   y
 16:  8d 04 02         lea   (%edx,%eax,1),%eax
 19:  5d               pop   %ebp
 1a:  c3               ret
```



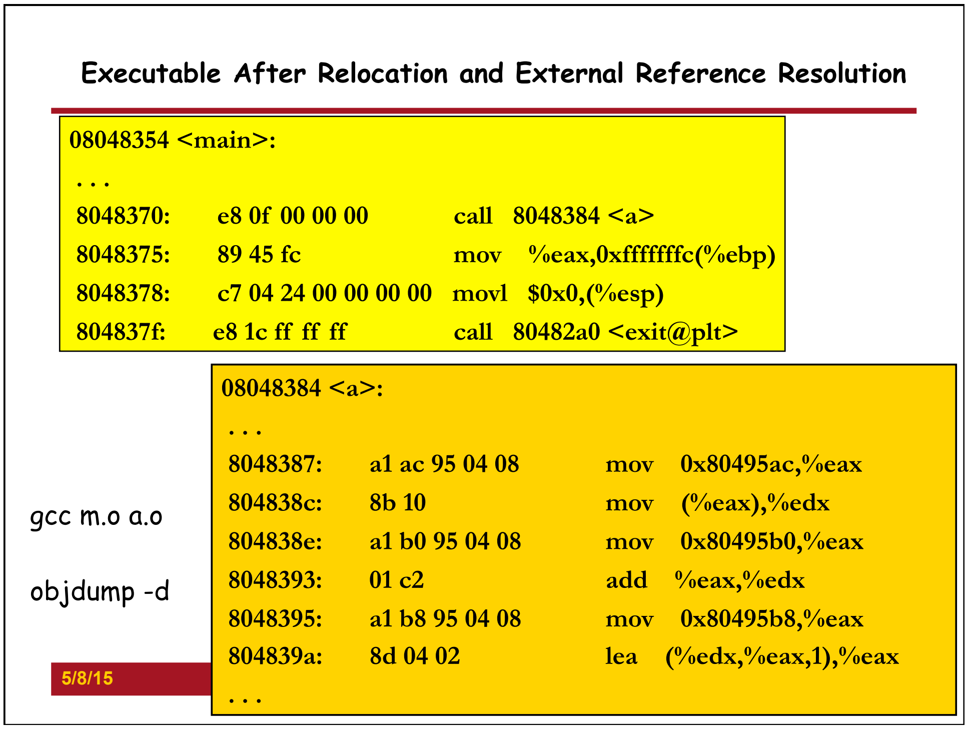
a.o Relocation Info (.data)

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
00000000 <ep>:
  0:  00 00            add   %al,(%eax)
  0:                   R_386_32   e
00000004 <x>:
  4:  0f 00 00         sldtl  (%eax)
    ...
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE         VALUE
00000004 R_386_32     ep
0000000b R_386_32     x
00000012 R_386_32     y
RELOCATION RECORDS FOR [.data]:
OFFSET   TYPE         VALUE
00000000 R_386_32     e
```

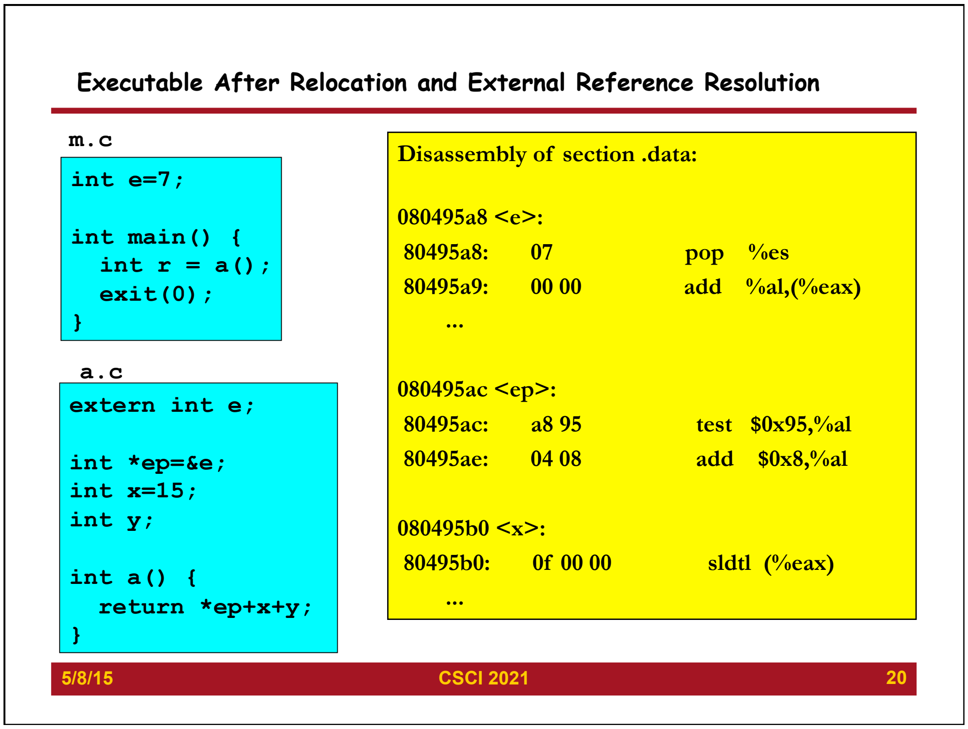


Executable After Relocation and External Reference Resolution

```
08048354 <main>:
 . . .
 8048370:     e8 0f 00 00 00          call   8048384 <a>
 8048375:     89 45 fc                mov    %eax,0xfffffffc(%ebp)
 8048378:     c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 804837f:     e8 1c ff ff ff          call   80482a0 <exit@plt>
```

```
08048384 <a>:
 . . .
 8048387:     a1 ac 95 04 08        mov    0x80495ac,%eax
 804838c:     8b 10                 mov    (%eax),%edx
 804838e:     a1 b0 95 04 08        mov    0x80495b0,%eax
 8048393:     01 c2                 add    %eax,%edx
 8048395:     a1 b8 95 04 08        mov    0x80495b8,%eax
 804839a:     8d 04 02              lea    (%edx,%eax,1),%eax
 . . .
```

```
gcc m.o a.o

objdump -d
```



Executable After Relocation and External Reference Resolution

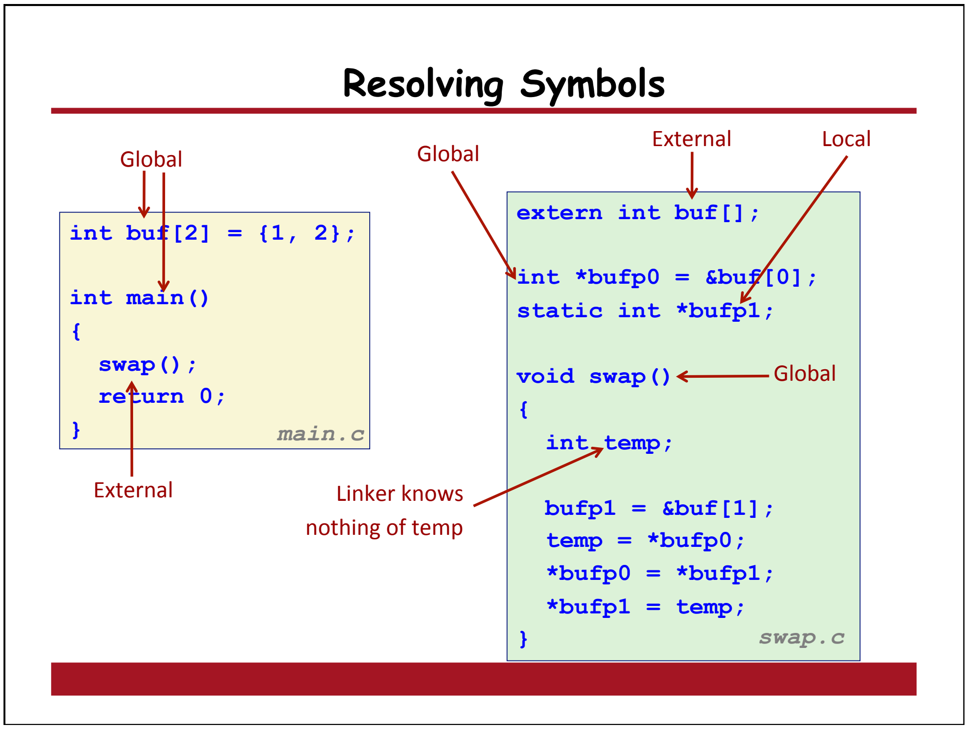
m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

```
Disassembly of section .data:

080495a8 <e>:
 80495a8:     07                pop    %es
 80495a9:     00 00             add    %al,(%eax)
       ...

080495ac <ep>:
 80495ac:     a8 95              test  $0x95,%al
 80495ae:     04 08              add   $0x8,%al

080495b0 <x>:
 80495b0:     0f 00 00            sldtl  (%eax)
       ...
```

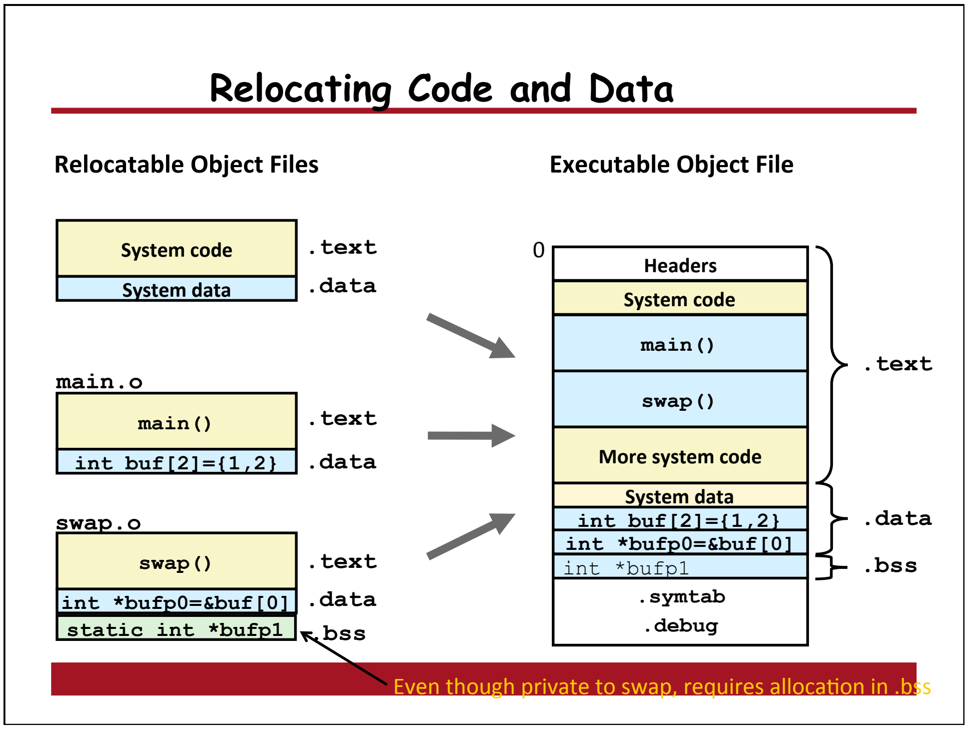


Resolving Symbols

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

int main()
{
   swap();
   return 0;
}                       main.c
```

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}                      swap.c
```

Global
External
Global
External
Local
Global
Linker knows nothing of temp



Relocating Code and Data

Relocatable Object Files

System code .text
System data .data

main.o
main() .text
int buf[2]={1,2} .data

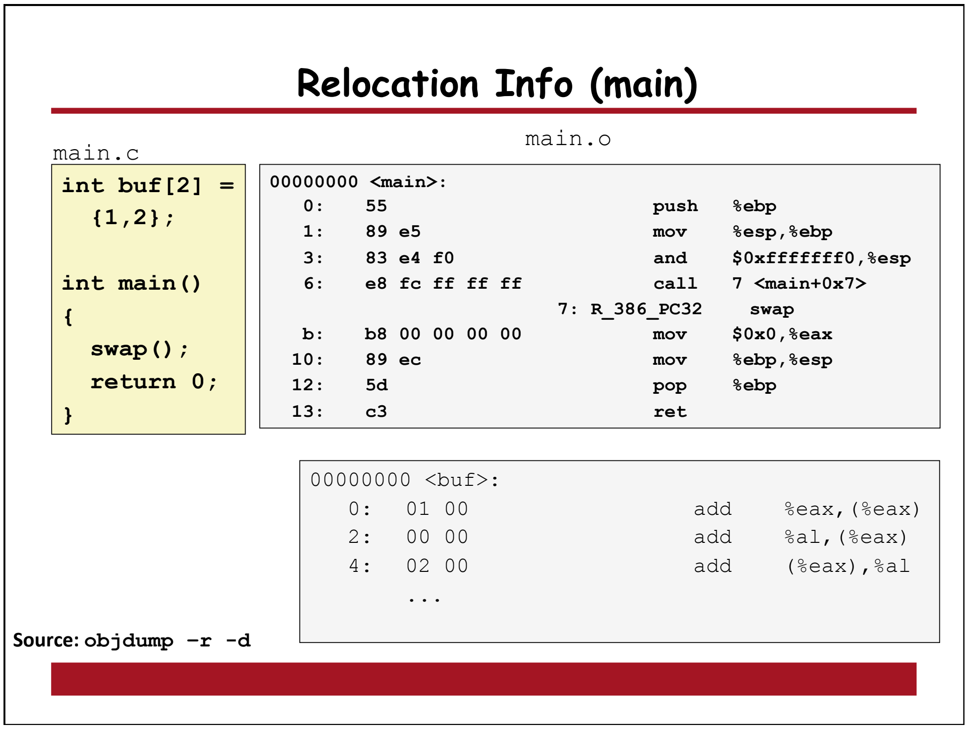
swap.o
swap() .text
int *bufp0=&buf[0] .data
static int *bufp1 .bss

Executable Object File

0
Headers
System code
main()
swap()
More system code
System data
int buf[2]={1,2}
int *bufp0=&buf[0]
int *bufp1
.symtab
.debug

.text
.data
.bss

Even though private to swap, requires allocation in .bss



Relocation Info (main)

main.c

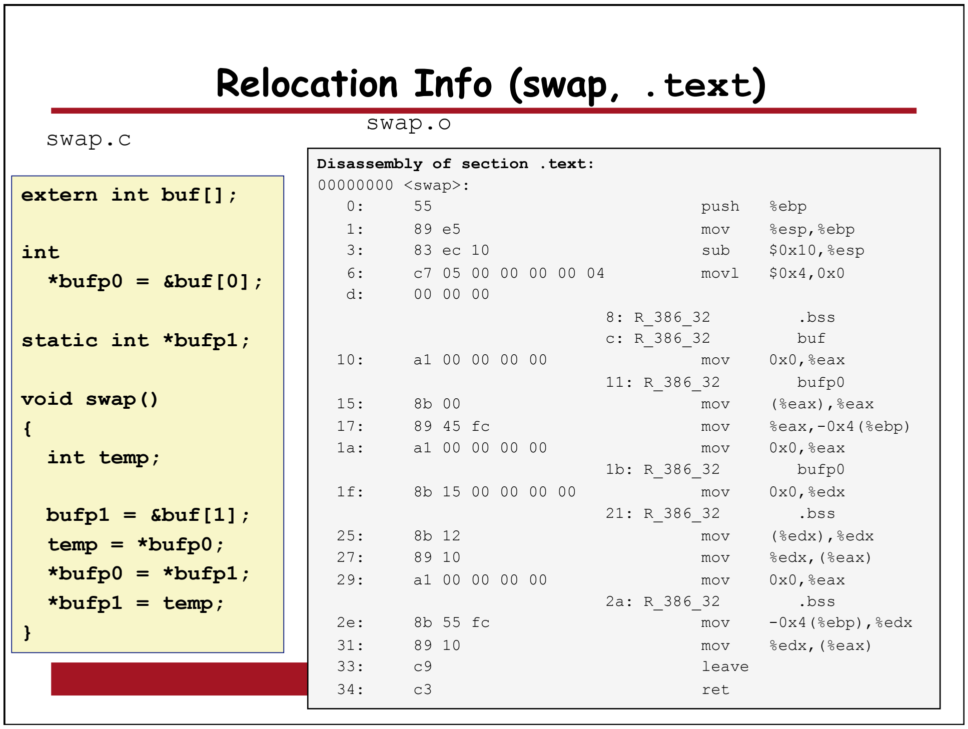
```
int buf[2] =
  {1,2};

int main()
{
  swap();
  return 0;
}
```

main.o

```
00000000 <main>:
   0:   55                          push   %ebp
   1:   89 e5                       mov    %esp,%ebp
   3:   83 e4 f0                    and    $0xfffffff0,%esp
   6:   e8 fc ff ff ff              call   7 <main+0x7>
                        7: R_386_PC32     swap
   b:   b8 00 00 00 00              mov    $0x0,%eax
  10:   89 ec                       mov    %ebp,%esp
  12:   5d                          pop    %ebp
  13:   c3                          ret
```

```
00000000 <buf>:
   0:   01 00                     add    %eax,(%eax)
   2:   00 00                     add    %al,(%eax)
   4:   02 00                     add    (%eax),%al
        ...
```

Source: `objdump -r -d`



Relocation Info (swap, .text)

swap.c

```
extern int buf[];

int
  *bufp0 = &buf[0];

static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

swap.o

```
Disassembly of section .text:
00000000 <swap>:
   0:     55                            push   %ebp
   1:     89 e5                         mov    %esp,%ebp
   3:     83 ec 10                      sub    $0x10,%esp
   6:     c7 05 00 00 00 00 04          movl   $0x4,0x0
   d:     00 00 00
                              8: R_386_32         .bss
                              c: R_386_32        buf
  10:     a1 00 00 00 00                mov    0x0,%eax
                              11: R_386_32       bufp0
  15:     8b 00                         mov    (%eax),%eax
  17:     89 45 fc                      mov    %eax,-0x4(%ebp)
  1a:     a1 00 00 00 00                mov    0x0,%eax
                              1b: R_386_32       bufp0
  1f:     8b 15 00 00 00 00             mov    0x0,%edx
                              21: R_386_32        .bss
  25:     8b 12                         mov    (%edx),%edx
  27:     89 10                         mov    %edx,(%eax)
  29:     a1 00 00 00 00                mov    0x0,%eax
                              2a: R_386_32        .bss
  2e:     8b 55 fc                      mov    -0x4(%ebp),%edx
  31:     89 10                         mov    %edx,(%eax)
  33:     c9                            leave
  34:     c3                            ret
```

Relocation Info (swap, .data)

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

```
Disassembly of section .data:

00000000 <bufp0>:
   0:   00 00           add    %al,(%eax)
        ...

Disassembly of section .bss:

00000000 <bufp1>:
   0:   00 00           add    %al,(%eax)
```

Executable Before/After Relocation (.text)

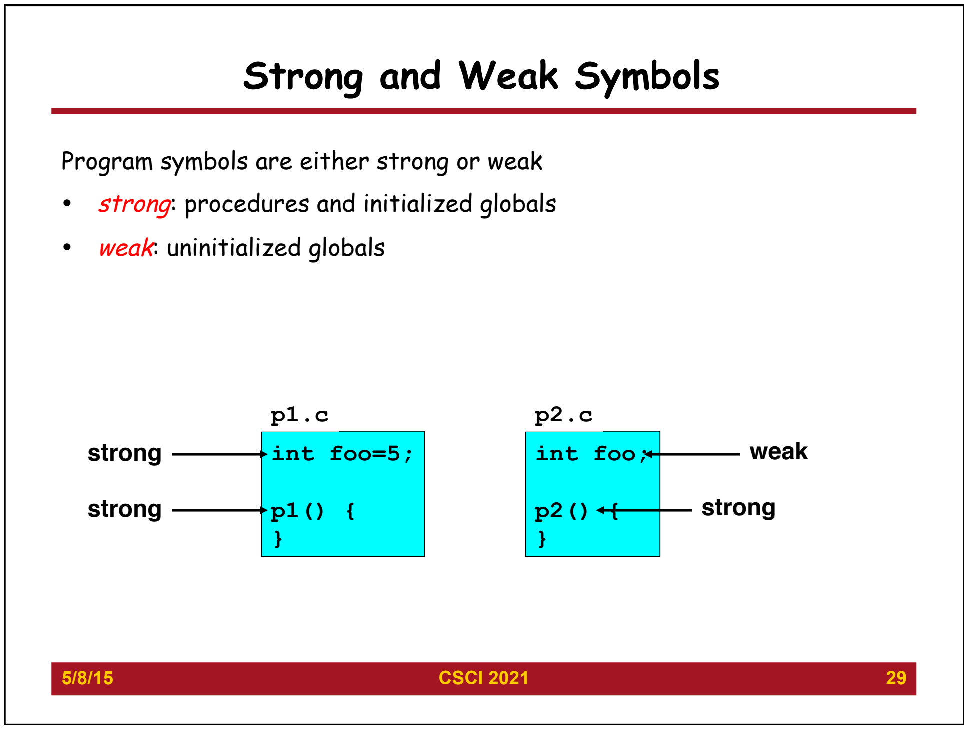
```
08048394 <main>:
 8048394:       55                        push   %ebp
 8048395:       89 e5                     mov    %esp,%ebp
 8048397:       83 e4 f0                  and    $0xfffffff0,%esp
 804839a:       e8 09 00 00 00            call   80483a8 <swap>
 804839f:       b8 00 00 00 00            mov    $0x0,%eax
 80483a4:       89 ec                     mov    %ebp,%esp
 80483a6:       5d                        pop    %ebp
 80483a7:       c3                        ret
```

Executable Before/After Relocation (.text)

```
80483a8:        55                          push   %ebp
80483a9:        89 e5                       mov    %esp,%ebp
80483ab:        83 ec 10                    sub    $0x10,%esp
80483ae:        c7 05 24 a0 04 08 14        movl   $0x804a014,0x804a024
80483b5:        a0 04 08
80483b8:        a1 18 a0 04 08              mov    0x804a018,%eax
80483bd:        8b 00                       mov    (%eax),%eax
80483bf:        89 45 fc                    mov    %eax,-0x4(%ebp)
80483c2:        a1 18 a0 04 08              mov    0x804a018,%eax
80483c7:        8b 15 24 a0 04 08           mov    0x804a024,%edx
80483cd:        8b 12                       mov    (%edx),%edx
80483cf:        89 10                       mov    %edx,(%eax)
80483d1:        a1 24 a0 04 08              mov    0x804a024,%eax
80483d6:        8b 55 fc                    mov    -0x4(%ebp),%edx
80483d9:        89 10                       mov    %edx,(%eax)
80483db:        c9                          leave
```

Executable After Relocation (.data)

```
Disassembly of section .data:
0804a010 <buf>:
 804a010:       01 00                 add    %eax,(%eax)
 804a012:       00 00                 add    %al,(%eax)
 804a014:       02 00                 add    (%eax),%al
        ...

0804a018 <bufp0>:
 804a018:       10                    .byte 0x10
 804a019:       a0                    .byte 0xa0
 804a01a:       04 08                 add    $0x8,%al

Disassembly of section .bss:
   …
0804a024 <bufp1>:
 804a024:       00 00                 add    %al,(%eax)
```



Strong and Weak Symbols

Program symbols are either strong or weak

- *strong*: procedures and initialized globals
- *weak*: uninitialized globals

p1.c

```
int foo=5;

p1() {
}
```

strong → int foo=5;

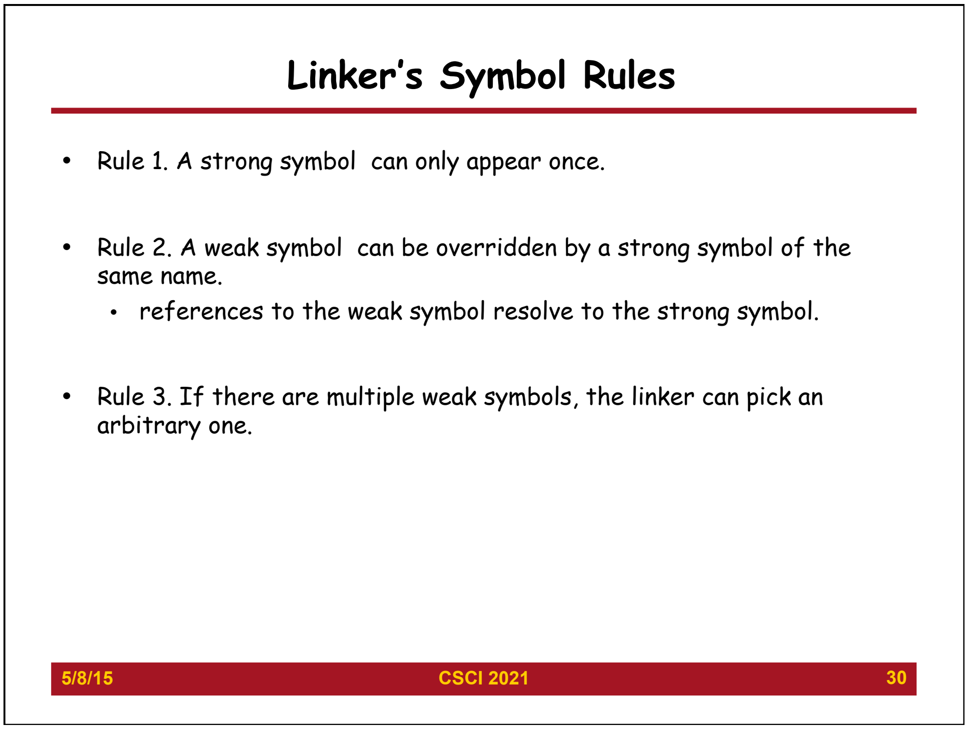
strong → p1() {

p2.c

```
int foo;

p2() {
}
```

weak → int foo;

strong → p2() {



Linker's Symbol Rules

- Rule 1. A strong symbol can only appear once.
- Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
  - references to the weak symbol resolve to the strong symbol.
- Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

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.

Role of .h Files

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

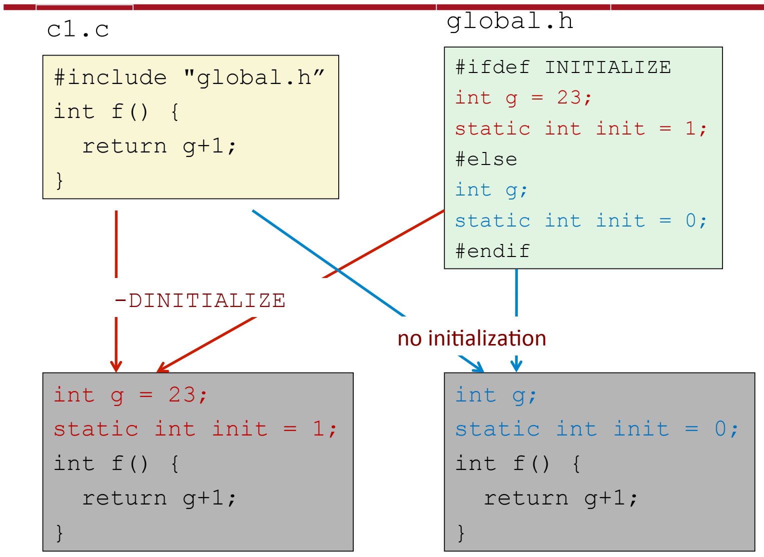
c1.c

```
#include "global.h"

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

c2.c

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

int main() {
  if (!init)
    g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Running Preprocessor



c1.c

```
#include "global.h”
int f() {
  return g+1;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

-DINITIALIZE

no initialization

```
int g = 23;
static int init = 1;
int f() {
  return g+1;
}
```

```
int g;
static int init = 0;
int f() {
  return g+1;
}
```

#include causes C preprocessor to insert file verbatim

Role of .h Files (cont.)

c1.c

```
#include "global.h"

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

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

c2.c

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

int main() {
  if (!init)
    g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

What happens:

```
gcc -o p c1.c c2.c
```

??

```
gcc -o p c1.c c2.c \
  -DINITIALIZE
```

??

Global Variables

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

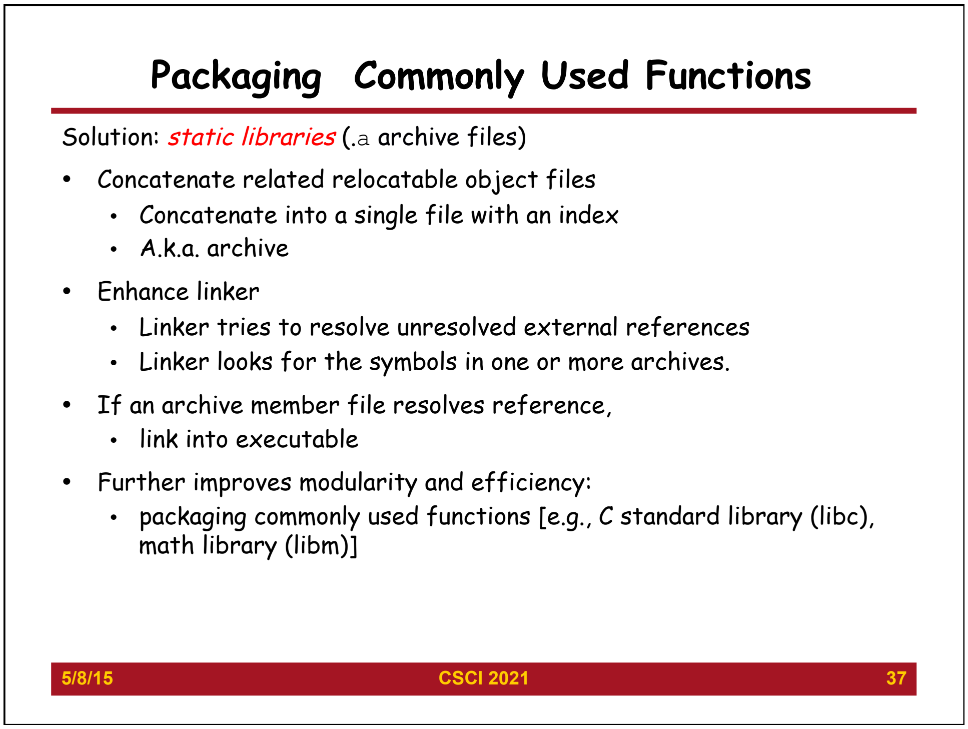
Packaging Commonly Used Functions

How to package functions commonly used by programmers?

- Math, I/O, memory management, string manipulation, etc.

Awkward, given the linker framework so far:

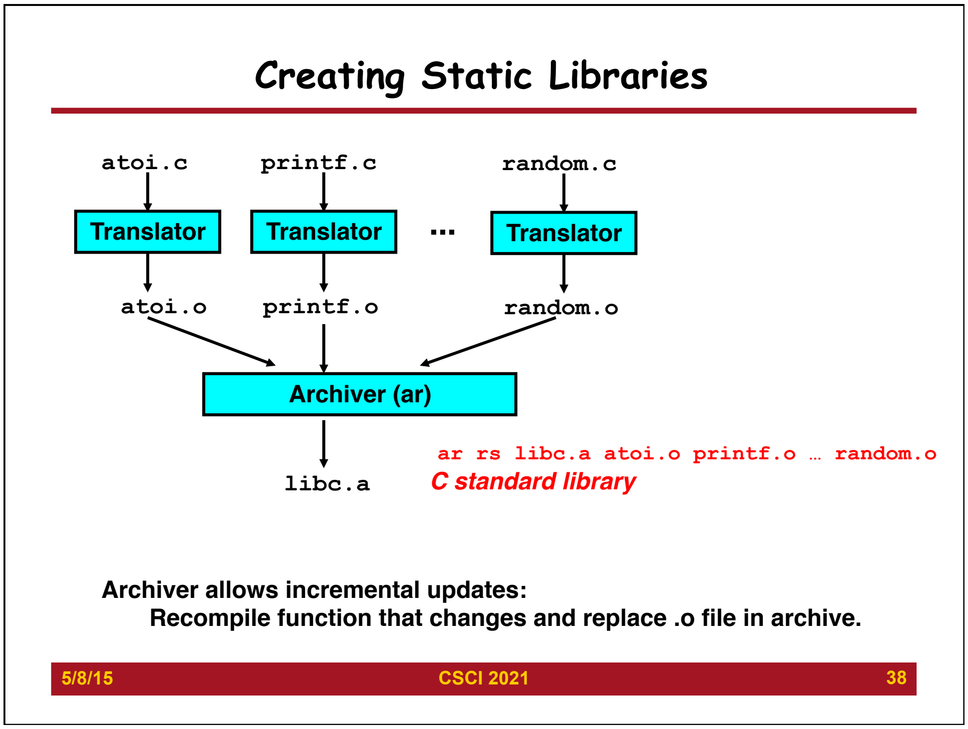
- Option 1: Put all functions in a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient
- Option 2: Put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome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



Packaging Commonly Used Functions

Solution: *static libraries* (`.a` archive files)

- Concatenate related relocatable object files
  - Concatenate into a single file with an index
  - A.k.a. archive
- Enhance linker
  - Linker tries to resolve unresolved external references
  - Linker looks for the symbols in one or more archives.
- If an archive member file resolves reference,
  - link into executable
- Further improves modularity and efficiency:
  - packaging commonly used functions [e.g., C standard library (libc), math library (libm)]



Creating Static Libraries

```
ar rs libc.a atoi.o printf.o … random.o
```

C standard library

Archiver allows incremental updates:
Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

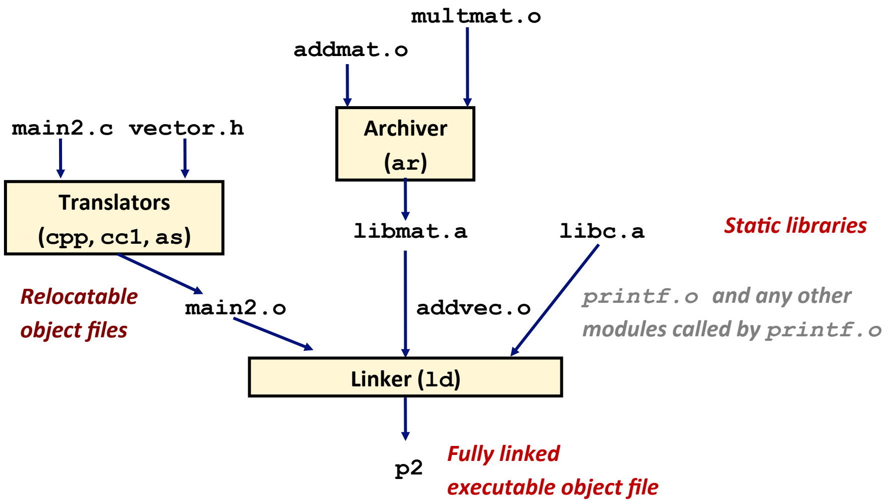
`libm.a` (the C math library)

- 1 MB archive of 226 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



Using Static Libraries

Linker's algorithm for resolving external references:

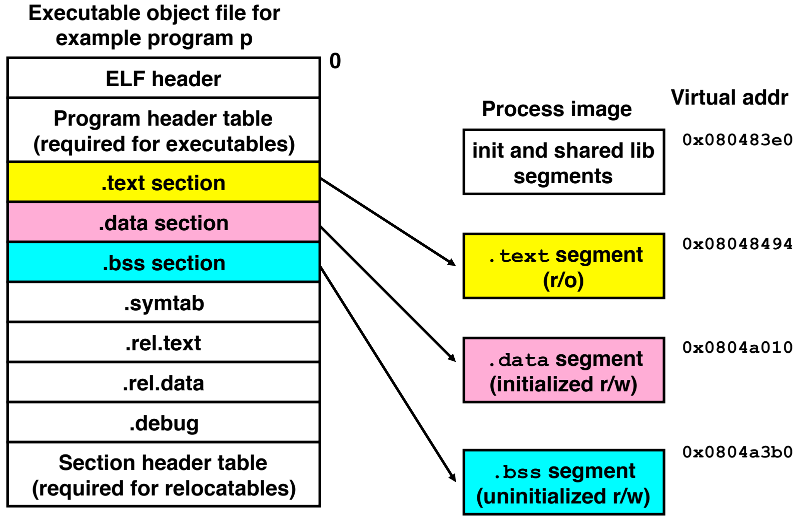
- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:

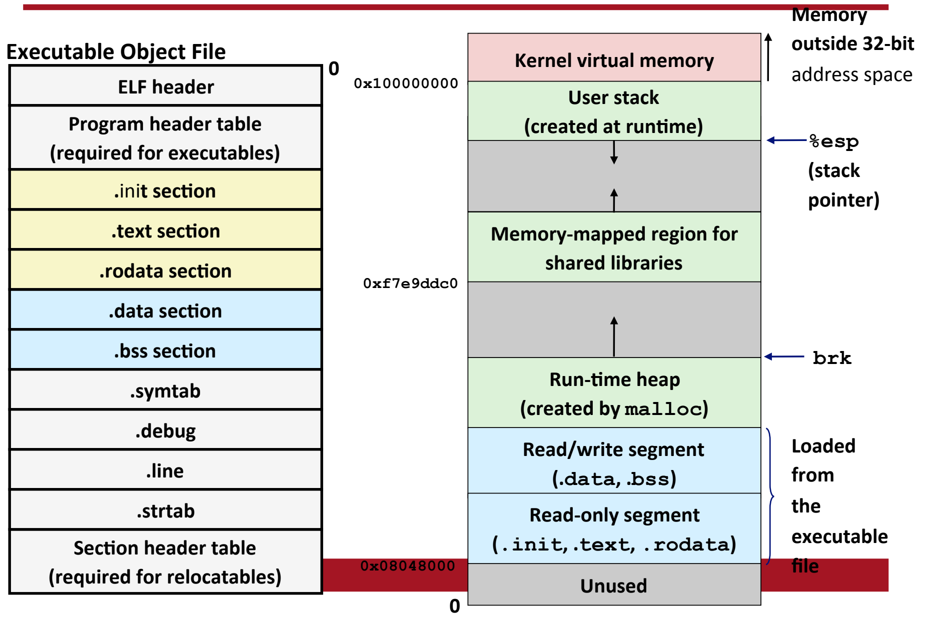
- Command line order matters!
- Moral: put libraries at the end of the command line.

```
lind40-14:/tmp> gcc -m32 -c main.c
lind40-14:/tmp> gcc -m32 -c swap.c
lind40-14:/tmp> ar -q libswap.a swap.o
lind40-14:/tmp> gcc -L. -m32 main.o -lswap
lind40-14:/tmp> gcc -L. -m32 -lswap main.o
main.o: In function `main':
main.c:(.text+0x7): undefined reference to `swap'
collect2: ld returned 1 exit status
```

Loading Executable Binaries


Executable object file for example program p
0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)
Process image
Virtual addr
init and shared lib segments
0x080483e0
.text segment (r/o)
0x08048494
.data segment (initialized r/w)
0x0804a010
.bss segment (uninitialized r/w)
0x0804a3b0

Loading Executable Object Files



Shared Libraries

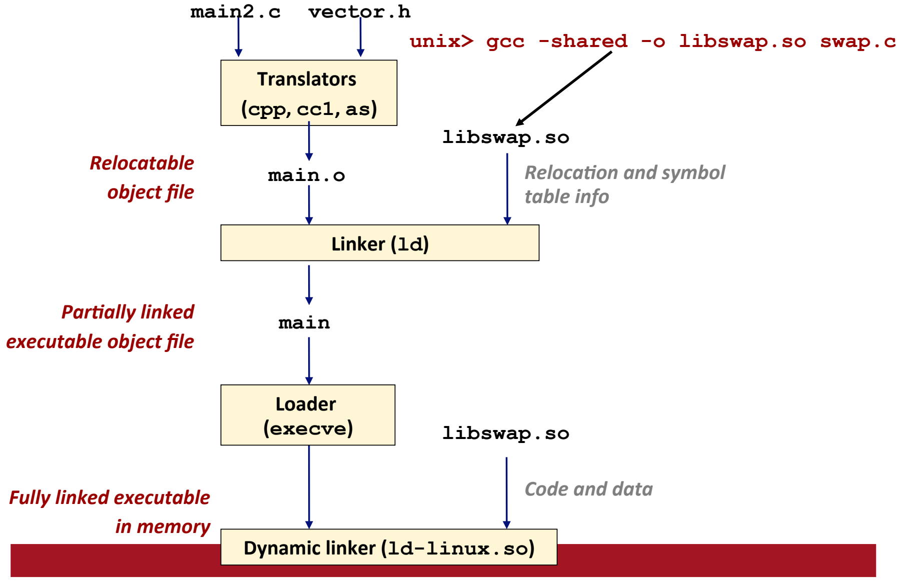
Static libraries have the following disadvantages:

- Potential for duplication
  - lots of common code in the executable files on a filesystem
  - e.g., every C program needs the standard C library
- Potential for duplication
  - lots of code in the virtual memory space of many processes.
- Relink
  - Minor bug fixes of system libraries require each application to explicitly relink

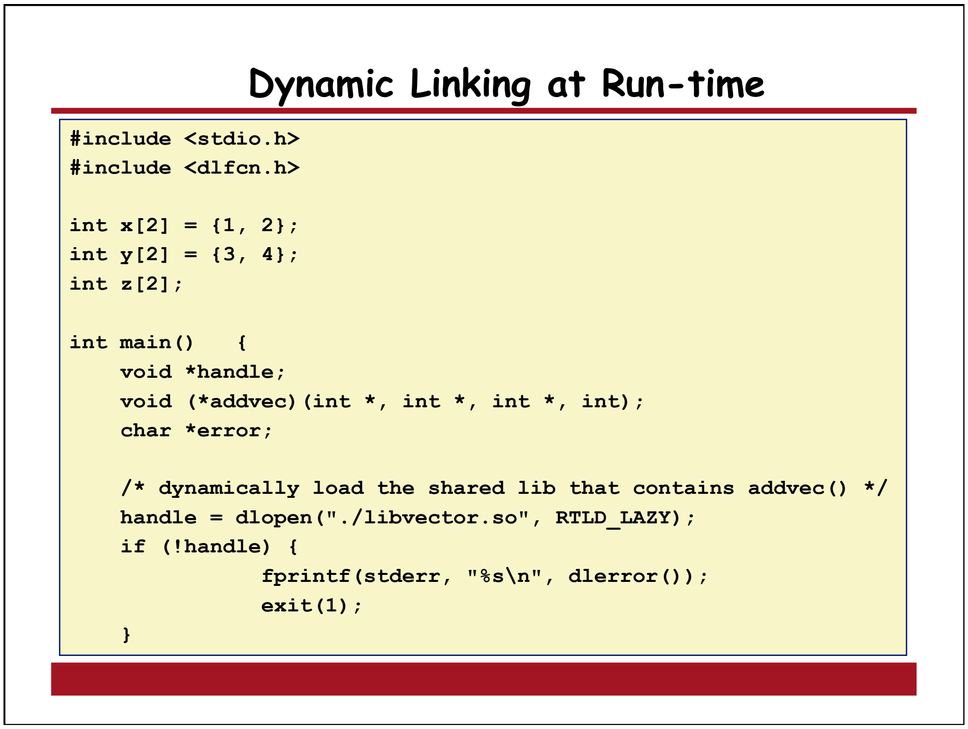
Shared Libraries (cont.)

- 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
- 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 memory

Dynamic Linking at Load-time

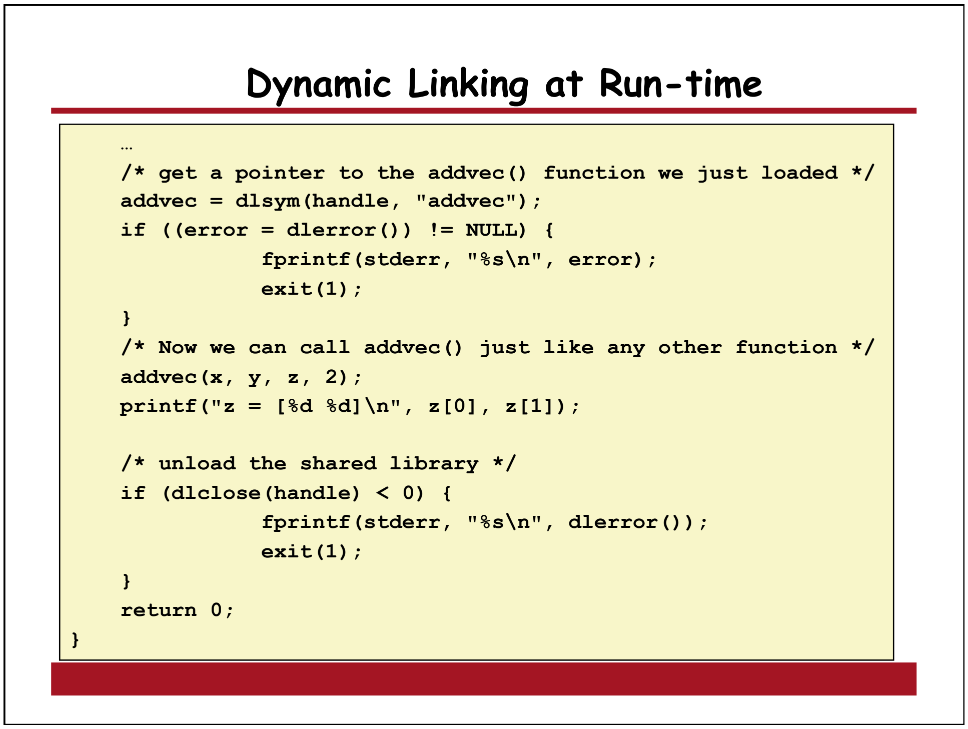


```
unix> gcc -shared -o libswap.so swap.c
```



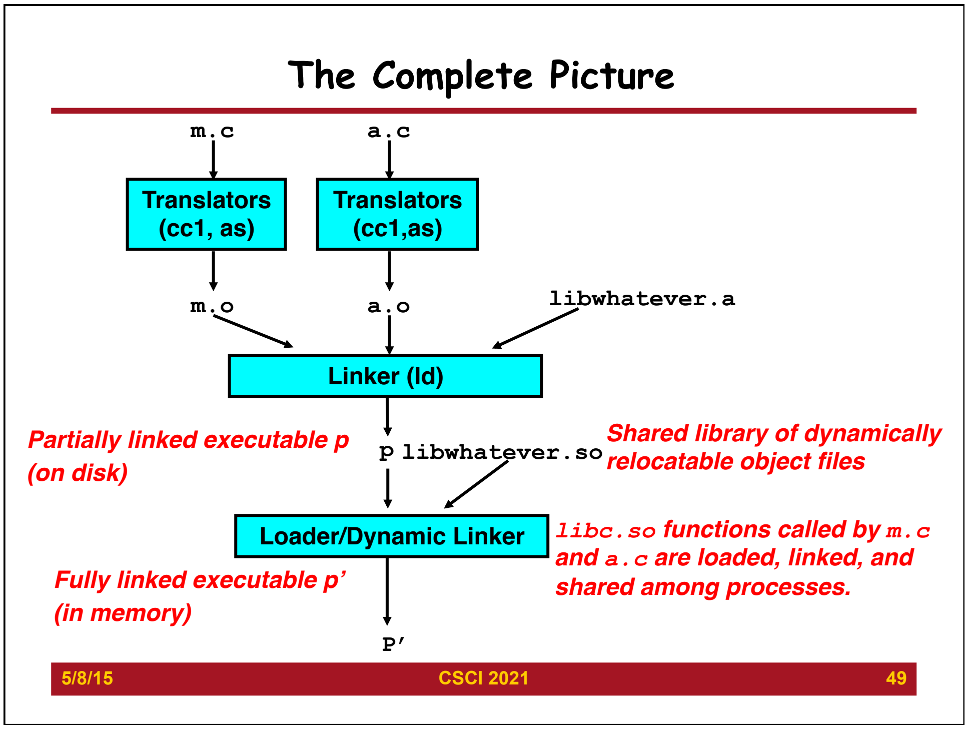
Dynamic Linking at Run-time

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

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

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

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



Dynamic Linking at Run-time

```
    …
    /* 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;
}
```



The Complete Picture

`m.c` `a.c`

Translators (cc1, as)

Translators (cc1,as)

`m.o` `a.o` `libwhatever.a`

Linker (ld)

Partially linked executable p (on disk)

`p` `libwhatever.so`

Shared library of dynamically relocatable object files

Loader/Dynamic Linker

`libc.so` functions called by `m.c` and `a.c` are loaded, linked, and shared among processes.

Fully linked executable p' (in memory)

`P'`

5/8/15 CSCI 2021 49