CS356 Unit 11

Linking

In complex C projects...

We would like to:

- **Split source into multiple .h / .c**
  - Should .c include .h? Before or after system libraries?
- **Compile these .h / .c units separately as .o files**
  - But only those that changed and their dependencies
- **Link these .o units into a single executable**
  - What if two .o define the same global variable/function?
  - How to use a variable or function defined by another .o?
  - How to keep a global variable or function “private”?
- **Save some .o in reusable libraries (.a / .so)**
  - How do we use their functions in .c files?
  - How do we find them during linking?

Why studying how linking works

- To better understand compiler/linking error messages
  - `main.c:(.text+0x13): undefined reference to `sum'`
- To understand how large programs are built
- To avoid subtle, hard-to-find bugs
- To understand OS & other system-level concepts
  - To help with CS 350!
- To exploit shared libraries (dynamic linkage)

Review

CS:APP 7.1

High Level Language Description

```
int func3(char str[])
{
  int i = 0;
  while(str[i] != 0) i++;
  return i;
}
```

.c/.cpp files

Preprocessor / Compiler

cpp/ cc1

```
func3:
  movl    $0, %eax
  jmp     .L2
.L3:
  addl    $1, %eax
.L2:
  movslq  %eax, %rdx
  cmpb    $0, (%rdi,%rdx)
  jne     .L3
  ret
```

Assembly (.asm/.s files)

Assembler

as

A "compiler" (i.e. gcc, clang) includes the assembler & linker

```
1110 0010 0101 1001
0110 1011 0000 1100
0100 1101 0111 1111
1010 1100 0010 1011
0001 0110 0011 1000
```

Object/Machine Code (.o files)

Linker

ld

```
1110 0010 0101 1001
0110 1011 0000 1100
0100 1101 0111 1111
1010 1100 0010 1011
0001 0110 0011 1000
```

Executable Binary Image

Loader / OS

Program Executing

A single .c file

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

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

int main()
{
    int val = sum(array, 2);
    return val;
}

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

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

Without the prototype, we would have to move the definition of sum before its use in main.

What about circular dependencies?

Splitting over **multiple .c**

(a) main.c — code/link/main.c

```
1   int sum(int *a, int n);
2
3   int array[2] = {1, 2};
4
5   int main()
6   {
7       int val = sum(array, 2);
8       return val;
9   }
```

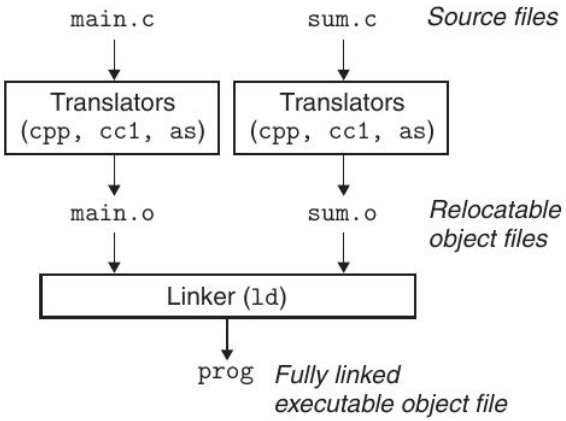
code/link/main.c

(b) sum.c — code/link/sum.c

```
1   int sum(int *a, int n)
2   {
3       int i, s = 0;
4
5       for (i = 0; i < n; i++) {
6           s += a[i];
7       }
8       return s;
9   }
```

code/link/sum.c

Figure 7.1 **Example program 1.** The example program consists of two source files, `main.c` and `sum.c`. The `main` function initializes an array of `int`s, and then calls the `sum` function to sum the array elements.



`gcc -Og -o prog main.c sum.c`

```
cpp [other arguments] main.c /tmp/main.i
cc1 /tmp/main.i -Og [other arguments] -o /tmp/main.s
as [other arguments] -o /tmp/main.o /tmp/main.s
ld -o prog [system objects] /tmp/main.o /tmp/sum.o
```

(same for sum.o)

Note: we are not using headers yet, and that can create bugs.

Compilation Units

- We want functions defined in one file to be able to be called in another
- But the compiler only compiles one file at a time…
  How does it know if the functions exist elsewhere?
  - It doesn't… it only checks when the linker runs (last step in compilation)
  - But it does require a prototype to verify & know the argument/return types

Q. If shuffle_test.c is compiled into a .o, how do we know the address of shuffle?

```
void shuffle(int *items, int len)
{
  /* code */
}
```

shuffle.c

```
int main()
{
  int cards[52];
  /* Initialize cards */
  ...
  // Shuffle cards
  shuffle(cards, 52);
  return 0;
}
```

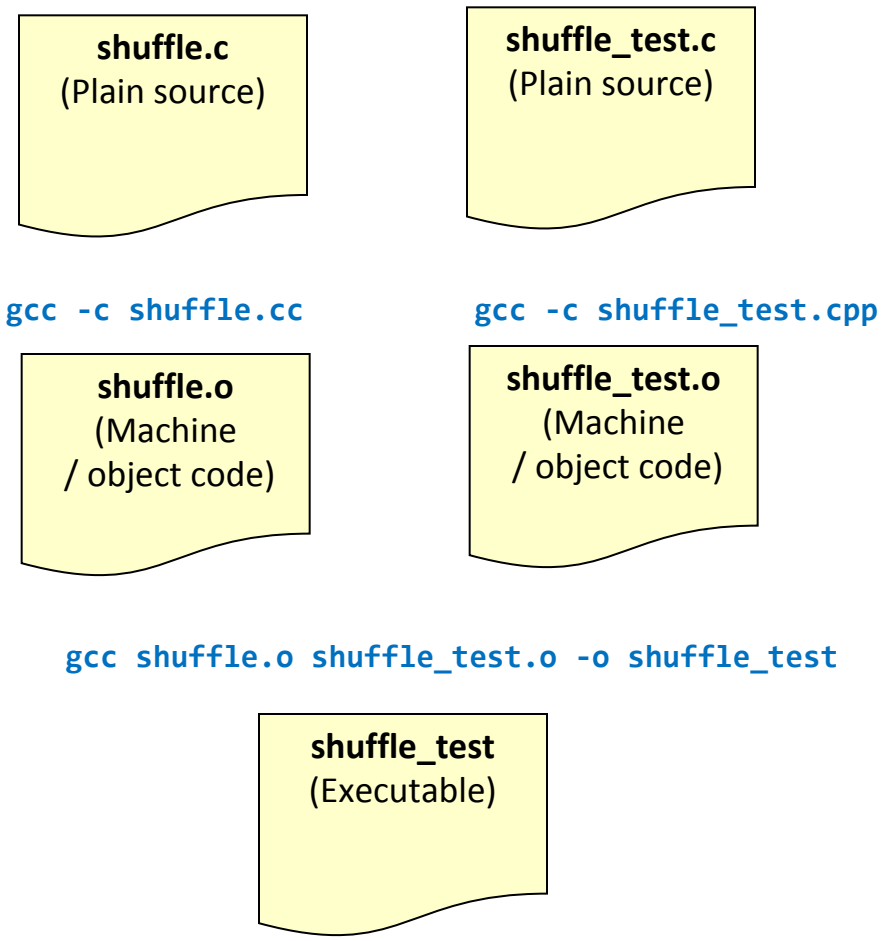
shuffle_test.c

```
void shuffle(int *items, int len);

int main()
{
  int cards[52];
  /* Initialize cards */
  ...

  shuffle(cards, 52);
  return 0;
}
```

shuffle_test.c

Linking

- After we compile to object code we eventually need to link all the files together and their function calls
- Without the -c, gcc will always try to link
- The linker will
  - **Verify referenced functions exists somewhere**
  - **Combine all the code & data together into an executable**
  - **Update the machine code to tie the references together**



static keyword

- In the context of C, the keyword 'static' in front of a global variable or function indicates the symbol is only visible within the **current compilation unit** and should not be visible (accessed) by other source code files
- Can be used as a sort of 'private' helper function declaration

```
struct Person { .. };

// Globals
int person_count = 0;
static int other_count = 0;

// Functions
void person_init(struct Person *p);
static void person_init_helper(
   struct Person* p);

// Definitions (code) for the
// functions
```

person.c

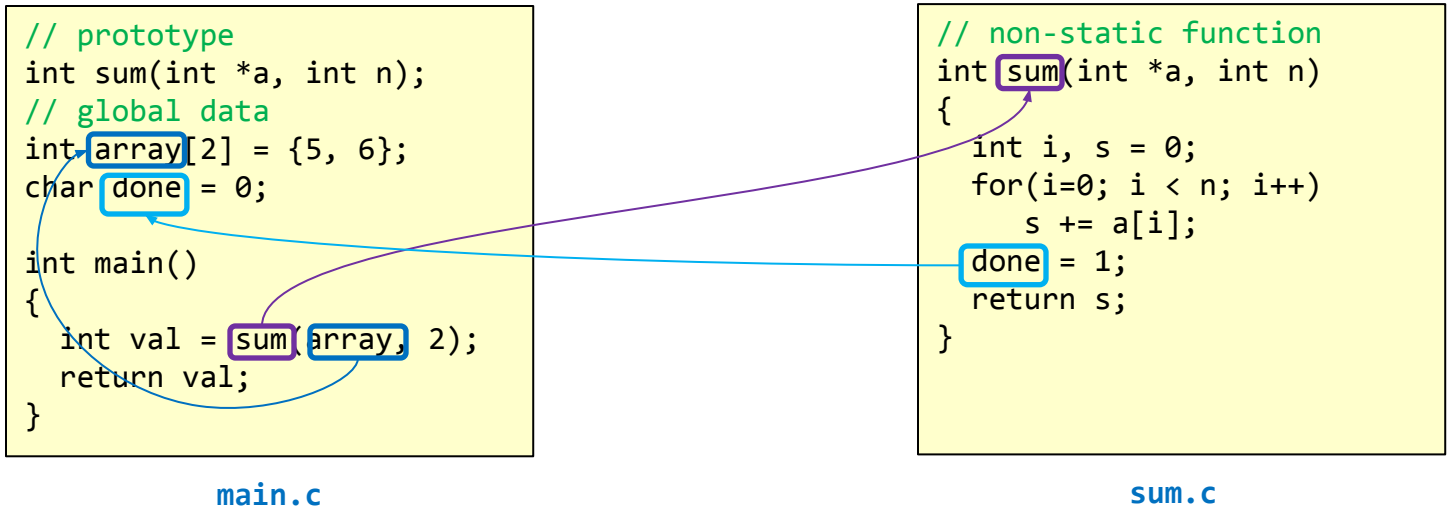
```
// these could come from a header person.h
struct Person { .. };
void person_init(struct Person*);
void person_init_helper(struct Person*);

int f1() {
   person_count++;        // Will compile
   other_count++;         // Will NOT compile

   struct Person p;

   person_init(&p);          // Will compile
   person_init_helper(&p);   // Will NOT
}
```

other.c

LINKING OVERVIEW

A First Look (1)

- Consider the example below:
  - Global variables: array and done
  - Functions: sum() and main()
- Linker needs to ensure the code references the appropriate memory locations for the code & data



A First Look (2)

- Each file can be compiled to object code separately
  - Notice the links are left blank (0) for now by the compiler

```
// prototype
int sum(int* a, int n);
// global data
int array[2] = {5, 6};
char done = 0;

int main()
{
  int val = sum(array, 2);
  return val;
}
```

$ gcc -O1 -c main.c

main.o

```
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
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

```
// non-static function
int sum(int* a, int n)
{
  int i, s = 0;
  for(i=0; i < n; i++)
     s += a[i];
  done = 1;
  return s;
}
```

$ gcc -O1 -c sum.c

sum.o

```
0000000000000000 <sum>:
   0:   85 f6                   test   %esi,%esi
   2:   7e 1d                   jle    21 <sum+0x21>
   4:   48 89 fa                mov    %rdi,%rdx
   7:   8d 46 ff                lea    -0x1(%rsi),%eax
   a:   48 8d 4c 87 04          lea    0x4(%rdi,%rax,4),%rcx
   f:   b8 00 00 00 00          mov    $0x0,%eax
  14:   03 02                   add    (%rdx),%eax
  16:   48 83 c2 04             add    $0x4,%rdx
  1a:   48 39 ca                cmp    %rcx,%rdx
  1d:   75 f5                   jne    14 <sum+0x14>
  1f:   eb 05                   jmp    26 <sum+0x26>
  21:   b8 00 00 00 00          mov    $0x0,%eax
  26:   c6 05 00 00 00 00 01    movb   $0x1,0x0(%rip)
  2d:   c3                      retq
```

A First Look (3)

- The linker will produce an executable with all references resolved to their exact addresses

```
// prototype
int sum(int *a, int n);
// global data
int array[2] = {5, 6};
char done = 0;

int main()
{
  int val = sum(array, 2);
  return val;
}
```

```
// non-static function
int sum(int *a, int n)
{
  int i, s = 0;
  for(i=0; i < n; i++)
     s += a[i];
  done = 1;
  return s;
}
```

main

```
00000000004004d6 <sum>:
  4004d6:       85 f6                   test   %esi,%esi
  4004d8:       7e 1d                   jle    4004f7 <sum+0x21>
  4004da:       48 89 fa                mov    %rdi,%rdx
  4004dd:       8d 46 ff                lea    -0x1(%rsi),%eax
  4004e0:       48 8d 4c 87 04          lea    0x4(%rdi,%rax,4),%rcx
  4004e5:       b8 00 00 00 00          mov    $0x0,%eax
  4004ea:       03 02                   add    (%rdx),%eax
  4004ec:       48 83 c2 04             add    $0x4,%rdx
  4004f0:       48 39 ca                cmp    %rcx,%rdx
  4004f3:       75 f5                   jne    4004ea <sum+0x14>
  4004f5:       eb 05                   jmp    4004fc <sum+0x26>
  4004f7:       b8 00 00 00 00          mov    $0x0,%eax
  4004fc:       c6 05 36 0b 20 00 01    movb   $0x1,0x200b36(%rip) # 601039 <done>
  400503:       c3                      retq

0000000000400504 <main>:
  400504:       48 83 ec 08             sub    $0x8,%rsp
  400508:       be 02 00 00 00          mov    $0x2,%esi
  40050d:       bf 30 10 60 00          mov    $0x601030,%edi
  400512:       e8 bf ff ff ff          callq  4004d6 <sum>
  400517:       48 83 c4 08             add    $0x8,%rsp
  40051b:       c3                      retq
```

```
$ gcc main.o sum.o -o main
```

Linker Tasks

CS:APP 7.2

- A linker has two primary tasks:
  - **Symbol resolution:** Resolve which single definition each symbol (function name, global variable, or static variable) resolves
  - **Relocation:** Associate a memory location to each symbol and then modifying all code references to that location
    - Object files start at offset 0 from their text/data sections; when linking all files must be placed into a single executable and code/data relocated

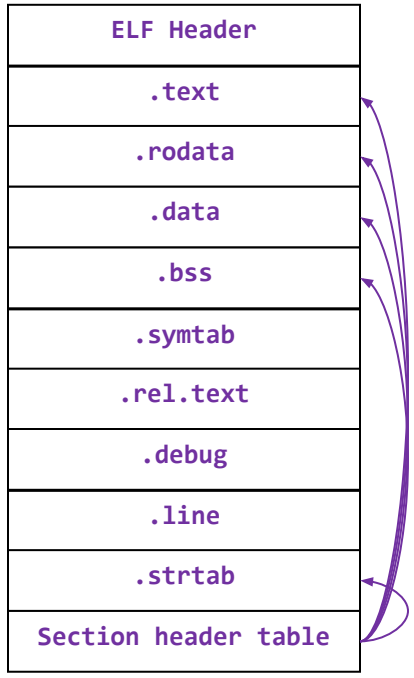
Object Files

CS:APP 7.3

- 3 kinds of object files:
  - **Relocatable object file** (typical .o file)
    Code/data along with book-keeping info for the linker
  - Executable object file (created by linker, can run it as ./prog)
    Binary that can be loaded into memory by the OS loader
  - Shared object file (.so file)
    Dynamically linked at load or run-time with some other executable
- Each OS defines its own format
  - Windows: Portable Executable (PE) format
  - Mac OS: Mach-O format
  - Linux/Unix: Executable & Linked Format (ELF)
    - We'll study this one

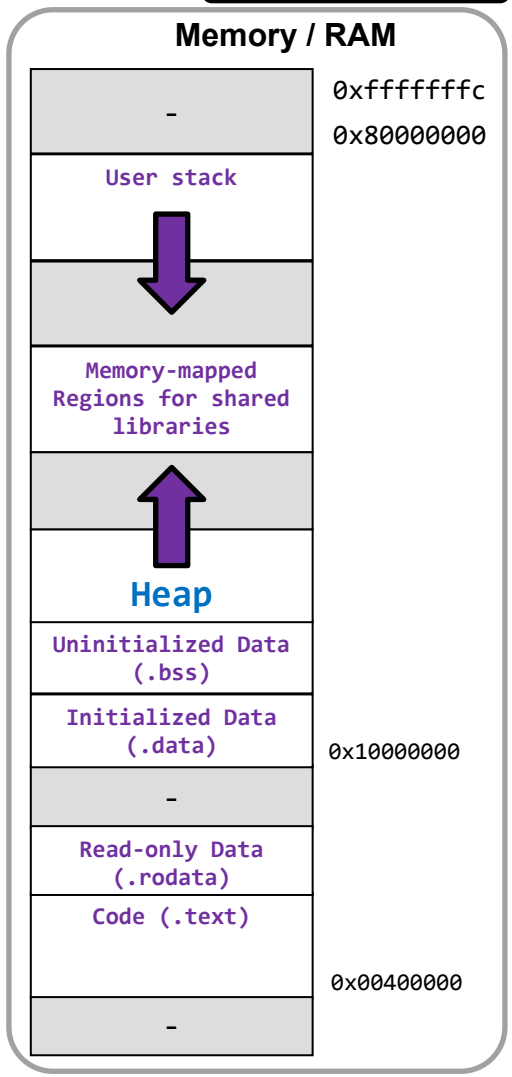
Relocatable Object ELF Sections

- Object files are made of various sections
  - Some map to actual memory sections when the program will execute
  - Others are for bookkeeping purposes to help the linker or debugger



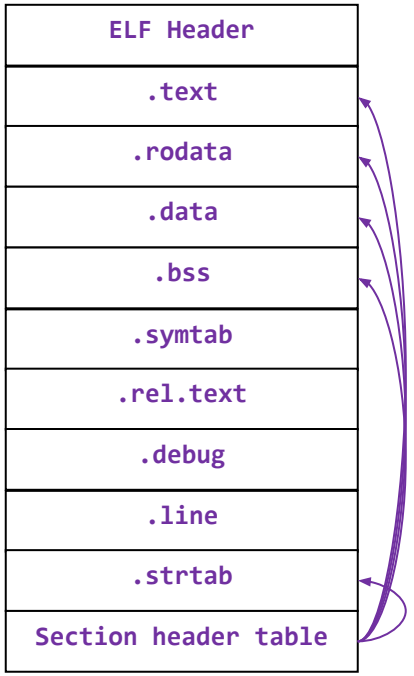
(Some sections will eventually map directly to memory sections of the executable while others are for bookkeeping purposes to help the linker or loader)

CS:APP 7.4



Section Descriptions

- **.text**: Machine **code** of instructions (executable code)
- **.rodata**: Read-only data (string constants, jump tables for switch)
- **.data**: **Initialized** global and static variables
- **.bss**: **Uninitialized** global and static variables (no actual space in .o file)
  - Will be zeroed by startup code when program is loaded
- .**symtab**: Symbol table with information about functions and global variables that are **defined or referenced** in this .o or executable
  - **Why no local variables?**
    They are on the stack and cannot be accessed by other code units
- **.rel.text & .rel.data**: locations in .text or .data section that reference outside functions or globals so that they later can be modified by the linker (only for .o)
- **.debug & .line**: A symbol table for locals and other definitions as well as line number to instruction mappings (included when the -g flag is used)
- **.strtab**: A string table of all the strings used in .symtab and .debug



Sample ELF-Header

STEP 1: SYMBOL RESOLUTION

Kinds of Symbols

CS:APP 7.5

- **Global symbols**
  - **Non-static**, global variables and functions that are defined in a compilation unit that can be referenced by other units
    - Function **definitions** and **global variables**
- **External global symbols**
  - Global symbols used in a compilation unit but undefined
    - Function **prototypes** or **global variables with `extern`**
- **Local symbols**
  - **Static** global variables and functions
  - NOT local variables (those are on the stack and the linker does not need to know about them)

Examples

```
// prototype
int sum(int* a, int n);
// global data
int array[2] = {5, 6};
char done = 0;

int main()
{
  int val = sum(array, 2);
  return val;
}
```

Global	array, done, main
External	sum
Local	
Linker-Ignored	val

```
#include <stdio.h>

int x=1, z=0;
static int y=5;

static int foo(int bar)
{
  x += bar;
  y--; z++;
  return x;
}

int main(int argc, char** argv)
{
  printf("%d\n", foo(3));
  return 0;
}
```

Global	x, z, main
External	printf
Local	foo, y
Linker-Ignored	argc, argv, bar

Symbol Table

- The compiler will store information about each symbol in the object file
- Fields
  - Value (relative offset in the section or absolute address of its location)
  - Bind ('local' = static definitions vs 'global')
  - Ndx (Section: 1=text, 3=data, 4=bss, UND=external)
  - Type ('object' = data, 'func' = function)

```
$ gcc -c res1.c
```

```c
#include <stdio.h>

int x=1, z=0;
static int y=5;

static int foo(int bar)
{
  x += bar;
  y--; z++;
  return x;
}

int main(int argc, char** argv)
{
  printf("%d\n", foo(3));
  return 0;
}
```

```
$ readelf -s res1.o
```

```
Symbol table '.symtab' contains 15 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS res1.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    3 y
     6: 0000000000000000    62 FUNC    LOCAL  DEFAULT    1 foo
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    11: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 x
    12: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    4 z
    13: 000000000000003e    49 FUNC    GLOBAL DEFAULT    1 main
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Duplicate Symbol Resolution Rules

CS:APP 7.6

- If duplicate 'local' symbols… **error**
- For global symbols, the compiler defines:
  - **Strong** symbols: **non-static functions** and **initialized non-static** global variables
  - **Weak** symbols: **uninitialized, non-static** global variables
- Global resolution rules:
  - Two or more 'strong' definitions of a symbol… **error**
  - One 'strong' and one or more 'weak' definitions of a symbol… **choose the 'strong' definition**
  - Many duplicate 'weak' symbols: arbitrarily choose one to be *the* definition
    - Can disable this arbitrary choice and generate an error with **`-fno-common`** option to `gcc`

```
// strong
int error = 0;

// weak
int val;
```

Duplicate Resolution Example

```
$ gcc res2a.c res2b.c -o res2
```

```c
// res2a.c
#include <stdio.h>

void doit(int *sum);

// better: extern int error
int error;
int val;

int main() {
  doit(&val);
  printf("%d\n", val);
  return 0;
}
```

```c
// res2b.c
#include <stdio.h>

int error = 0;
int val;

void doit(int *sum) {
  int x, y;
  if(2 != scanf("%d %d", &x, &y)){
    error = 1;
    return;
  }
  *sum = x+y;
}
// would generate an error, if added
// int main() { return 0; }
```

Strong	main
Weak	error, val

Strong	doit, error
Weak	val

```
$ gcc -fno-common res2a.c res2b.c
/tmp/ccwo7BuS.o:(.bss+0x0): multiple definition of `error'
/tmp/ccbFljfF.o:(.bss+0x0): first defined here
/tmp/ccwo7BuS.o:(.bss+0x4): multiple definition of `val'
/tmp/ccbFljfF.o:(.bss+0x4): first defined here
collect2: error: ld returned 1 exit status
```

Duplicate Resolution Example

```
$ gcc res3a.c res3b.c -o res3
```

```c
// res3a.c
#include <stdio.h>

void doit(int *sum);

int val;
int val2;

int main()
{
  val = 1;
  doit();
  val++;
  printf("val=%d\n", val);
  return 0;
}
```

```c
// res3b.c
#include <stdio.h>

double val;


void doit(int *sum)
{
  int x;
  scanf("%d", &x);
  val += x;
}
```

Strong	main
Weak	val, val2

Strong	doit
Weak	val

```
$ gcc -fno-common res3a.c res3b.c
/tmp/ccHUiJtU.o:(.bss+0x0): multiple definition of `val'
/tmp/ccDcaoUE.o:(.bss+0x0): first defined here
/usr/bin/ld: Warning: size of symbol `val' changed from 4 in
/tmp/ccDcaoUE.o to 8 in /tmp/ccHUiJtU.o
collect2: error: ld returned 1 exit status
```

Global Variable Summary

- **Don't use them, if you can help it**
- If you must
  - Use `static`, if you can
  - If you define a global variable, **initialize it** (so that it will be a strong symbol and the compiler will catch duplicates)
  - If you reference a global variable from another module, use the `extern` keyword
    - Hopefully, provided by the **header of that module**!

When external types don't match

```c
/* main.c */
#include <stdio.h>
int z = 0x11223344;
void swap(int *x, int *y);

int main() {
    int x = 0x11223344;
    int y = 0x55667788;
    printf("z = %x\n", z);
    swap(&x, &y);
    printf("x = %x\n", x);
    printf("y = %x\n", y);
    printf("z = %x\n", z);
}
```

```c
/* swap.c */
short z;

void swap(short *x, short *y) {
    z = 0;
    short tmp = *x;
    *x = *y;
    *y = tmp;
}
```

```
$ gcc -Wall -Wextra -std=c99 main.c swap.c -o prog
$ ./prog
z = 11223344
x = 11227788
y = 55663344
z = 11220000
```

Compiling with `-flto`

```c
/* main.c */
#include <stdio.h>
int z = 0x11223344;
void swap(int *x, int *y);

int main() {
    int x = 0x11223344;
    int y = 0x55667788;
    printf("z = %x\n", z);
    swap(&x, &y);
    printf("x = %x\n", x);
    printf("y = %x\n", y);
    printf("z = %x\n", z);
}
```

```c
/* swap.c */
short z;

void swap(short *x, short *y) {
    z = 0;
    short tmp = *x;
    *x = *y;
    *y = tmp;
}
```

```
$ gcc -Wall -Wextra -std=c99 -flto main.c swap.c -o prog
main.c:4:6: warning: type of ‘swap’ does not match original declaration
[..]
swap.c:1:7: warning: type of ‘z’ does not match original declaration
[..]
main.c:3:5: note: type ‘int’ should match type ‘short int’
```

Using Headers

```c
/* main.h */
#ifndef MAIN_H
#define MAIN_H

extern int z;

#endif /* MAIN_H */
```

```c
/* swap.h */
#ifndef SWAP_H
#define SWAP_H

void swap(int *x, int *y);

#endif /* SWAP_H */
```

```
$ gcc -Wall -Wextra -std=c99 \
  main.c swap.c -o prog
$ ./prog
z = 11223344
x = 55667788
y = 11223344
z = 0
```

```c
/* main.c */
#include "main.h"
#include "swap.h"
#include <stdio.h>
int z = 0x11223344;
int main() {
    int x = 0x11223344;
    int y = 0x55667788;
    printf("z = %x\n", z);
    swap(&x, &y);
    printf("x = %x\n", x);
    printf("y = %x\n", y);
    printf("z = %x\n", z);
}
```

```c
/* swap.c */
#include "swap.h"
#include "main.h"

void swap(int *x, int *y) {
    z = 0;
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Strategy: Each unit includes its own prototypes/declarations.

- Non-matching types now result in **compile errors** within a unit
- Header guards are used to avoid double (or recursive) inclusion of headers
- Adding `int z = 42` to swap.c results in a **linking error**

STEP 2: RELOCATION

Executable Object File

CS:APP 7.8

- When the linker runs it can create an executable by combining all the object files to resolve all symbols, decide where all code and data will be placed in memory, and then relocating all references

main.o

```
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
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

sum.o

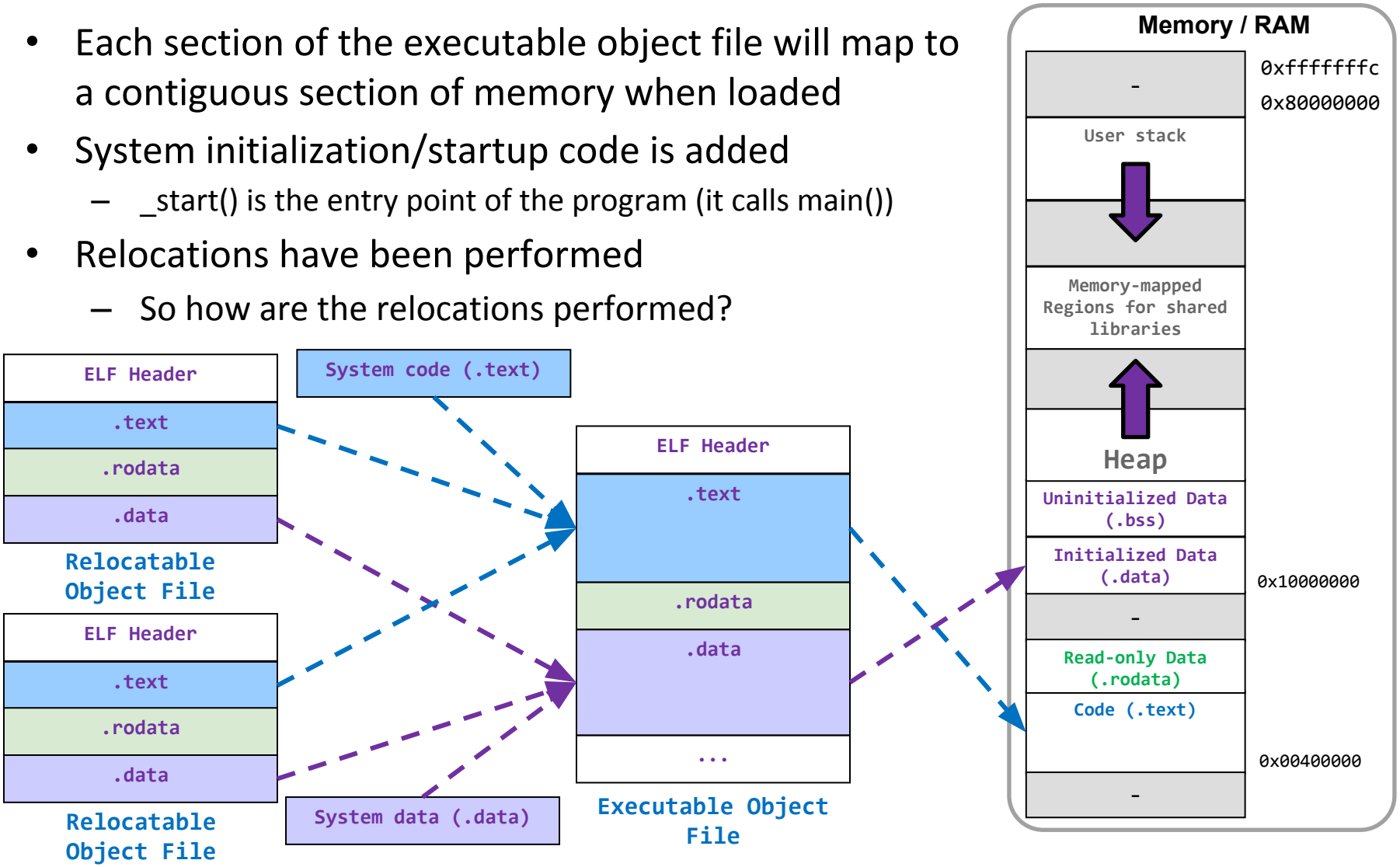
```
0000000000000000 <sum>:
   0:   85 f6                   test   %esi,%esi
...
  26:   c6 05 00 00 00 00 01    movb   $0x1,0x0(%rip)
  2d:   c3                      retq
```

Executable Object

```
00000000004004d6 <sum>:
  4004d6:       85 f6                   test   %esi,%esi
  ...
  4004fc:       c6 05 36 0b 20 00 01    movb   $0x1,0x200b36(%rip) # 601039 <done>
  400503:       c3                      retq

0000000000400504 <main>:
  400504:       48 83 ec 08             sub    $0x8,%rsp
  400508:       be 02 00 00 00          mov    $0x2,%esi
  40050d:       bf 30 10 60 00          mov    $0x601030,%edi
  400512:       e8 bf ff ff ff          callq  4004d6 <sum>
  400517:       48 83 c4 08             add    $0x8,%rsp
  40051b:       c3                      retq
```

Executable Object File

- Each section of the executable object file will map to a contiguous section of memory when loaded
- System initialization/startup code is added
  - _start() is the entry point of the program (it calls main())
- Relocations have been performed
  - So how are the relocations performed?



Relocation Review

CS:APP 7.7

- Object files left links to global symbols blank

```
// prototype
int sum(int* a, int n);
// global data
int array[2] = {5, 6};
char done = 0;

int main()
{
  int val = sum(array, 2);
  return val;
}
```

$ gcc -O1 -c main.c

main.o

```
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
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

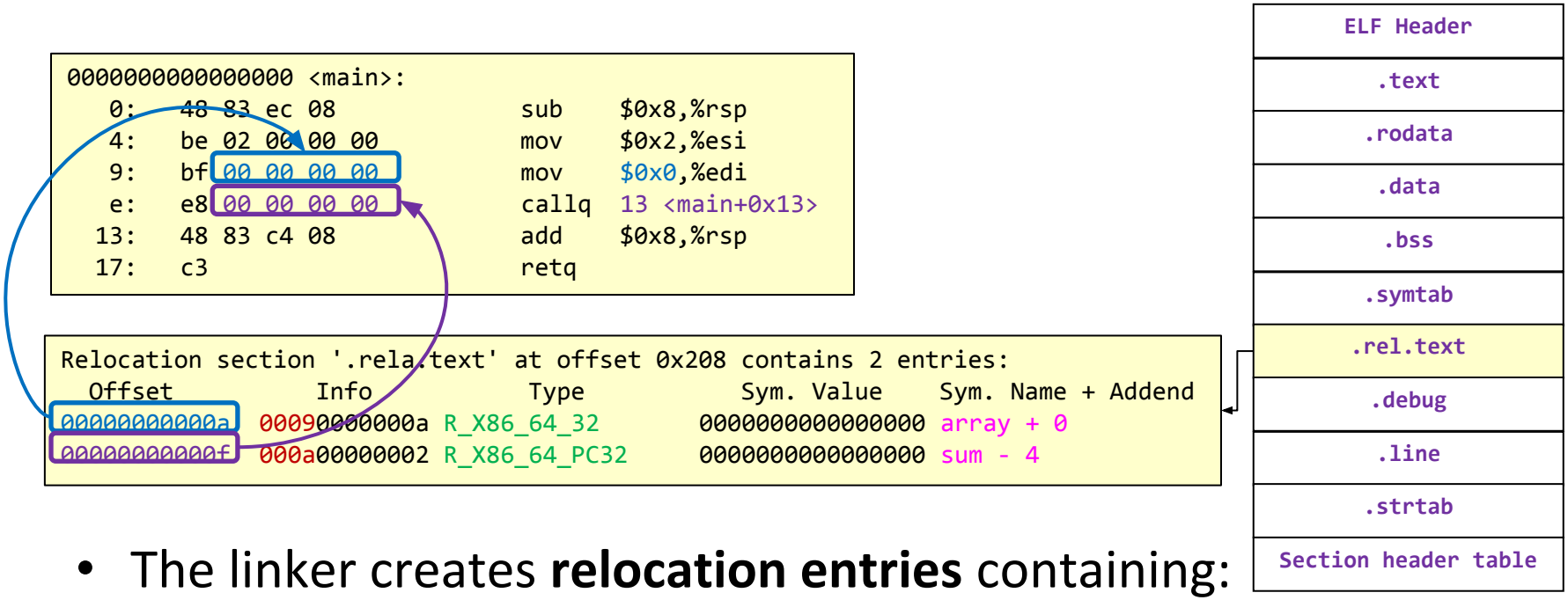
```
// non-static function
int sum(int* a, int n)
{
  int i, s = 0;
  for(i=0; i < n; i++)
     s += a[i];
  done = 1;
  return s;
}
```

$ gcc -O1 -c sum.c

sum.o

```
0000000000000000 <sum>:
   0:   85 f6                   test   %esi,%esi
   2:   7e 1d                   jle    21 <sum+0x21>
   4:   48 89 fa                mov    %rdi,%rdx
   7:   8d 46 ff                lea    -0x1(%rsi),%eax
   a:   48 8d 4c 87 04          lea    0x4(%rdi,%rax,4),%rcx
   f:   b8 00 00 00 00          mov    $0x0,%eax
  14:   03 02                   add    (%rdx),%eax
  16:   48 83 c2 04             add    $0x4,%rdx
  1a:   48 39 ca                cmp    %rcx,%rdx
  1d:   75 f5                   jne    14 <sum+0x14>
  1f:   eb 05                   jmp    26 <sum+0x26>
  21:   b8 00 00 00 00          mov    $0x0,%eax
  26:   c6 05 00 00 00 00 01    movb   $0x1,0x0(%rip)
  2d:   c3                      retq
```

Relocation Entries



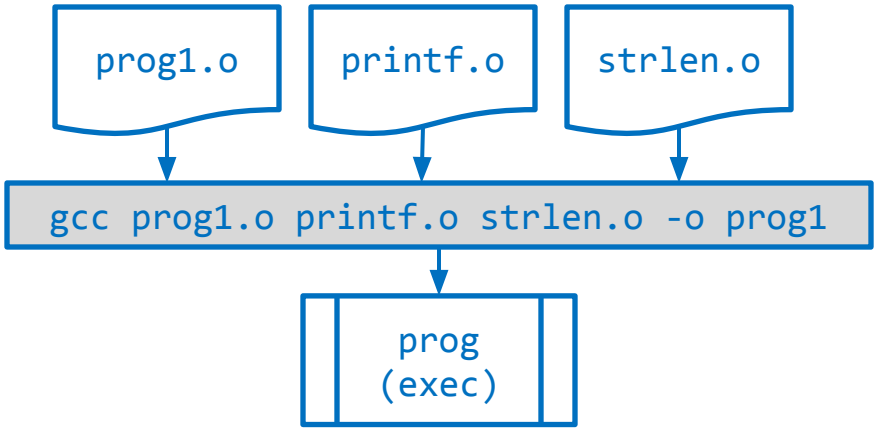
- The linker creates **relocation entries** containing:
  - Type of offset
    - **PC-Relative** (R_X86_64_PC32) or **Absolute Address** (R_X86_64_32)
  - Section offset where the relocation should be replaced
  - Symbol being referenced (and its symbol table index)

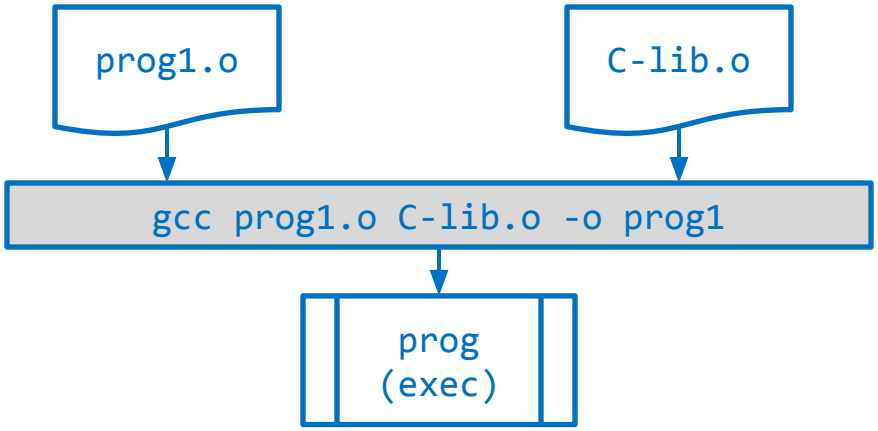
STATIC LIBRARIES

Need for Libraries

CS:APP 7.6

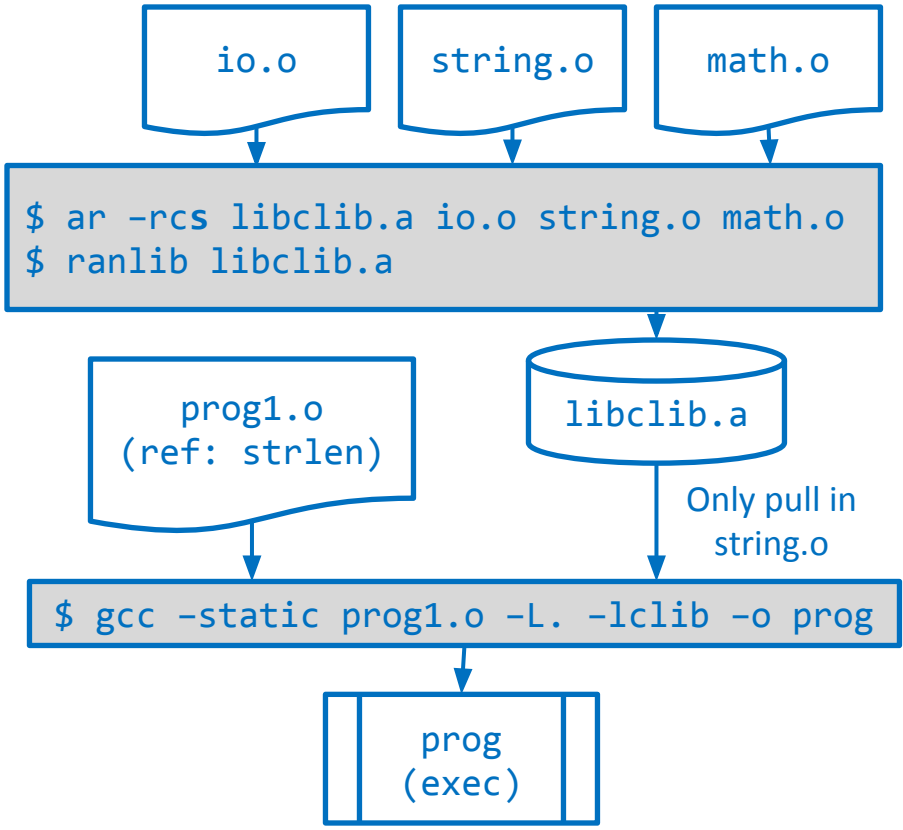
- Often have many related files containing commonly reused code (functions)
  - Think of the C library (I/O functions, string library, math, etc.) or related classes (and all the code of their member functions)
  - How should we compile and link these in?
- Option 1: Put each function in a separate file
  - Painful to track and write which files are needed
- Option 2: Put all code in a single file
  - Lot of unneeded, extra code to link in





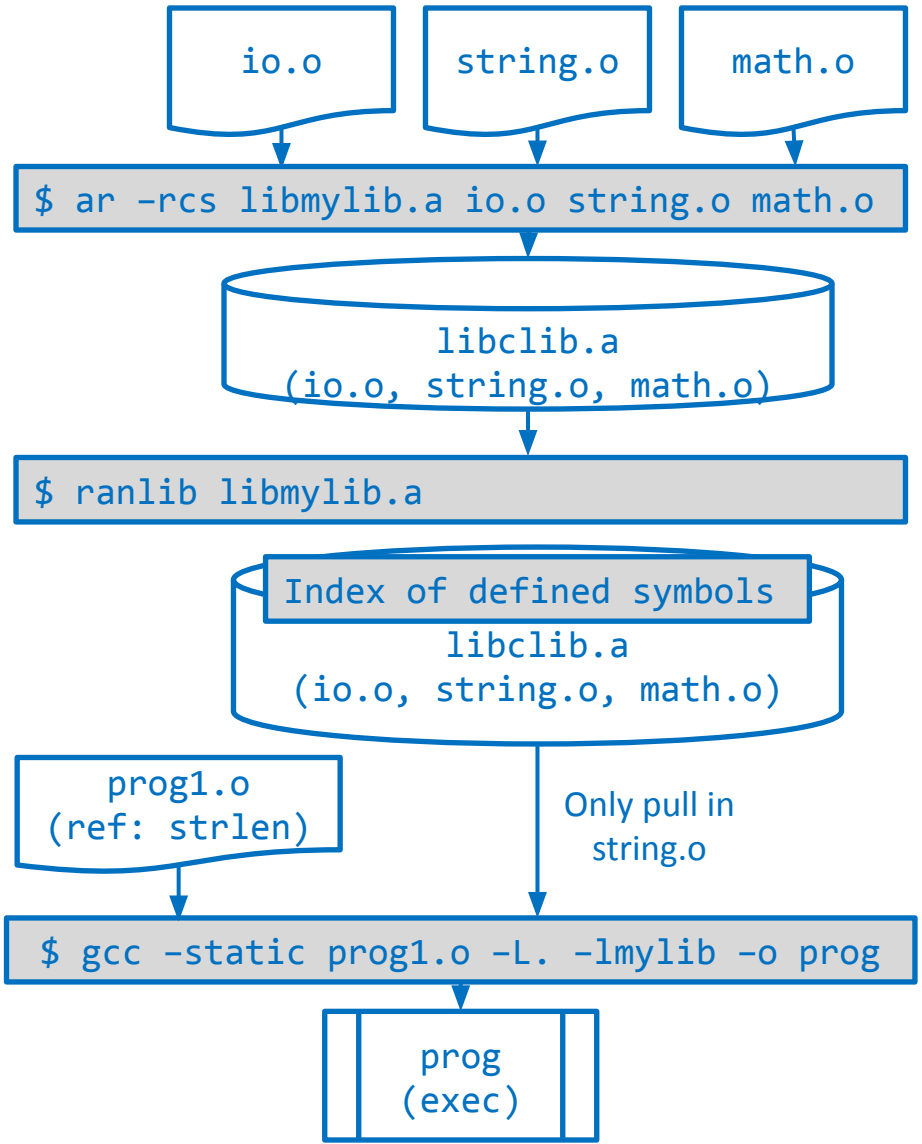
Benefits of a Library

- Compile all, possible object files and put them together into a library (archive) file
  - Linux: Archive (.a) file
  - Windows:
    Portable Executable (.lib) file
- When a program needs functions/data from the library, the compiler can include just the ones that are needed (by checking what undefined symbols from the program are defined in the library)



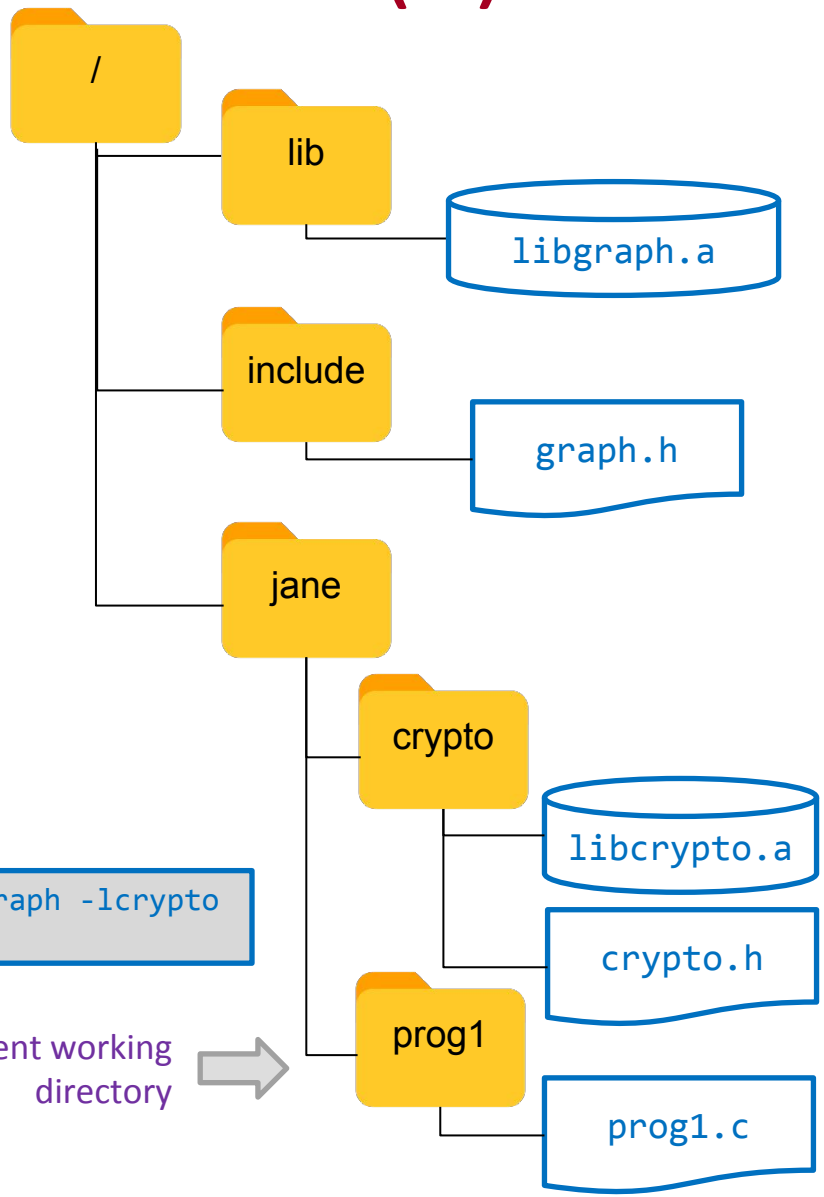
Static Libraries on Linux (1)

- ar (archiver) packs multiple object files into one
  - **Output file should start with "lib" prefix**
- ranlib adds an index of the symbols defined in the object files that are part of the library to enable quick determination of which object files to link in
- To link in code from a library use the **`-static`** option
  - **`-L`** indicates the path to folders to search for libraries
  - **`-l`** indicates the name of the library to link against (**without the "lib" prefix** since it assumes the file will have the "lib" prefix)



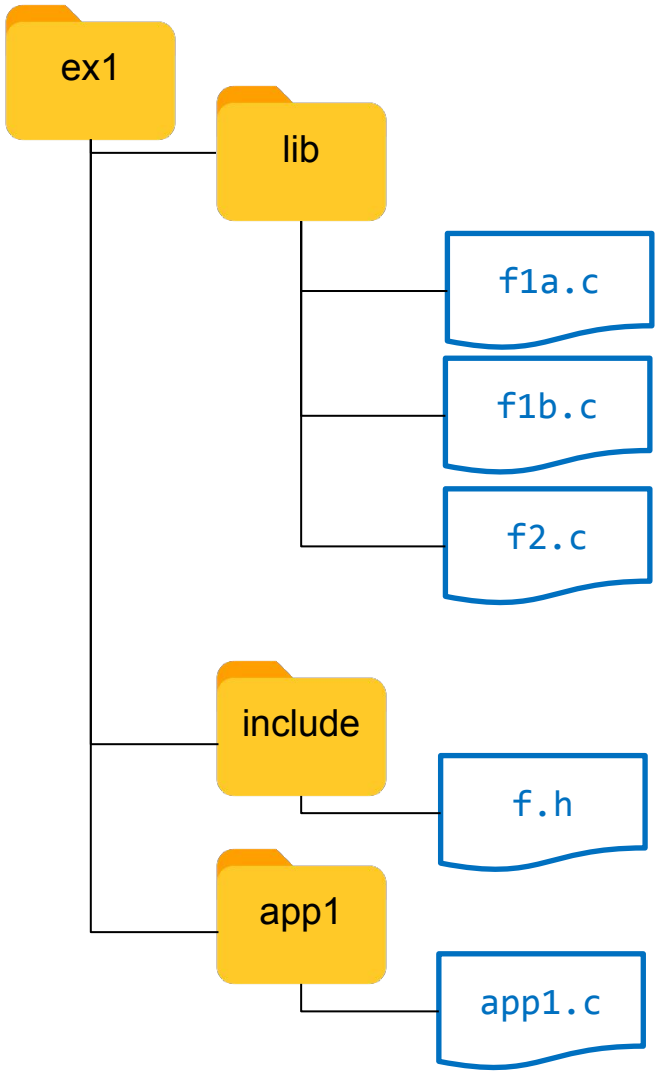
Static Libraries on Linux (2)

- `-I`*`include-path`* sets include search path for headers included by the source code being compiled
  - Can be relative or absolute
- `-L`*`lib-path`* sets search paths for libraries to be linked
  - Can be relative or absolute
- `-l`*`libname`* specifies the library to link into the code
  - Do not use the 'lib' prefix of the filename




```
$ gcc prog1.c -I/include -I../crypto -L../crypto -L/lib -lgraph -lcrypto
                                                              -o prog
```

Library Example




```c
int x;
int f11()
{
  x = 11;
  return x;
}
int f12()
{
  x = 12;
  return x;
}
```
f1a.c

```c
int x;
int f11()
{
  x = 1111;
  return x;
}
int f12()
{
  x = 1212;
  return x;
}
```
f1b.c

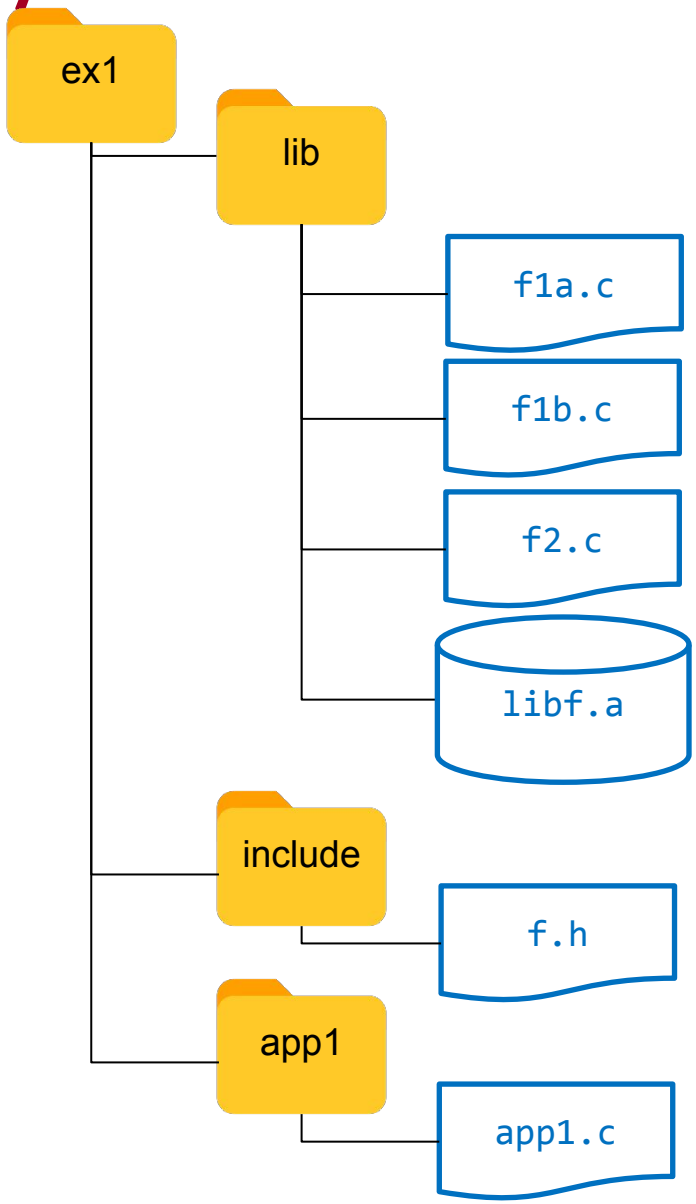
```c
#include "f.h"
#include <stdio.h>

int main()
{
  int t = f11();
  printf("%d\n",t);
  return 0;
}
```
app1.c

```c
int f11();
int f12();
int f21();
int f22();
```
f.h

```c
int y;
int f21()
{
  y = 21;
  return y;
}
int f22()
{
  y = 22;
  return y;
}
```
f2.c

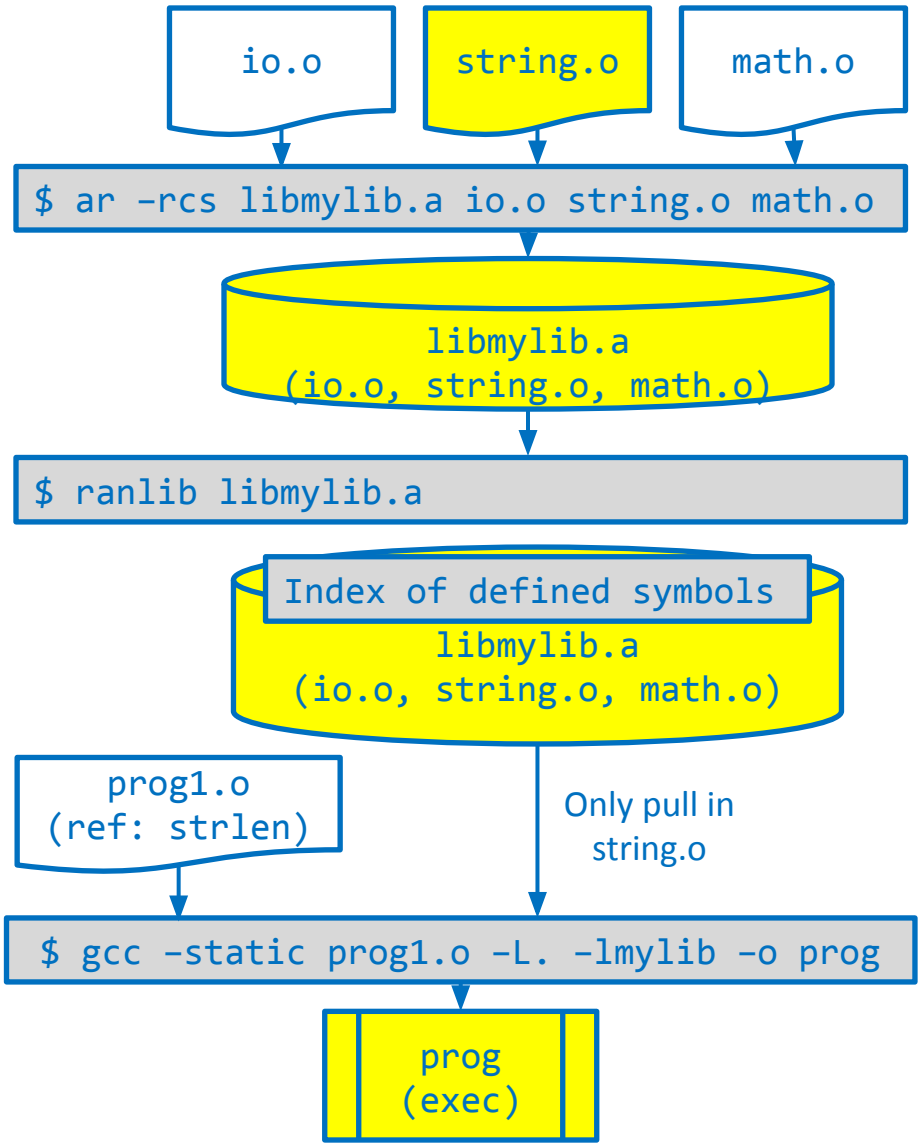
Try it! Static Library Exercise

```
cd lib
rm libf.a
gcc -c f1a.c f2.c
ar -rcs libf.a f1a.o f2.o
ranlib libf.a
cd ../app1
gcc app1.c            # fatal error: no 'f.h'
gcc -I../include app1.c # notice 'undefined reference
# need to link in the library
gcc -I../include app1.c -L../lib -lf
./a.out       # should see '11' output
objdump -d ./a.out > a.s
subl a.s &    # notice no f21/f22 functions
cd ../lib
rm libf.a
gcc -c f1b.c
ar -rcs libf.a f1b.o f2.o
ranlib libf.a
cd ../app1
./a.out       # notice no update in output
# need to recompile app1
gcc -I../include app1.c -L../lib -lf
./a.out
# ok now we get the update
cd ..
```



Static Library Issues

- What happens if we need to change the code in the library?
  - Need to **re-link all executables** that used the library
- What if multiple running programs linked against the library
  - Multiple copies of the code in memory, space wasted
- Is there a better way?
  - Shared libraries / dynamic linking



Shared objects (**.so**) and Dynamically Linked Libraries (**.dll**)

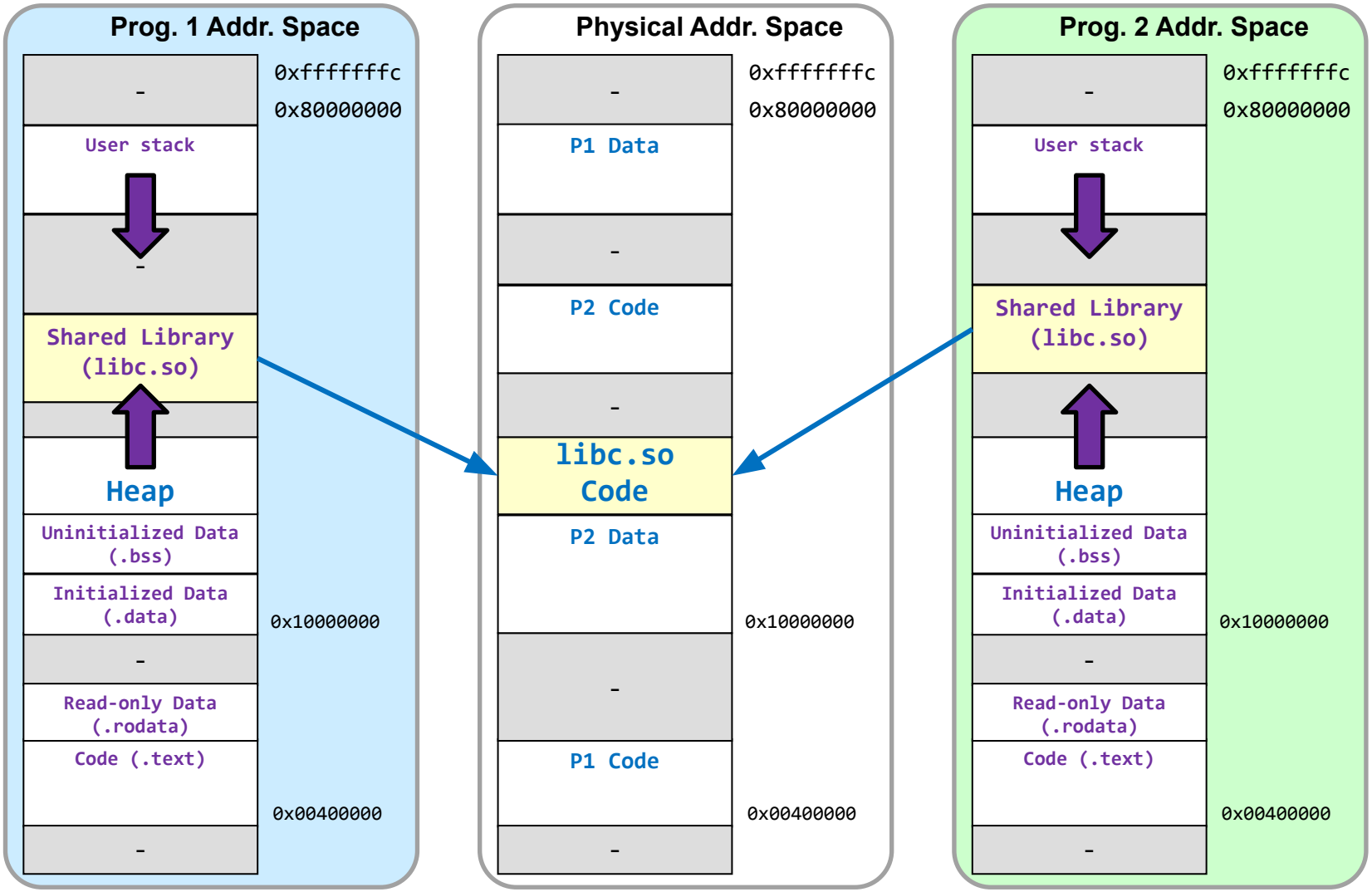
DYNAMIC/SHARED LIBRARIES

Shared Library Motivation

CS:APP 7.10

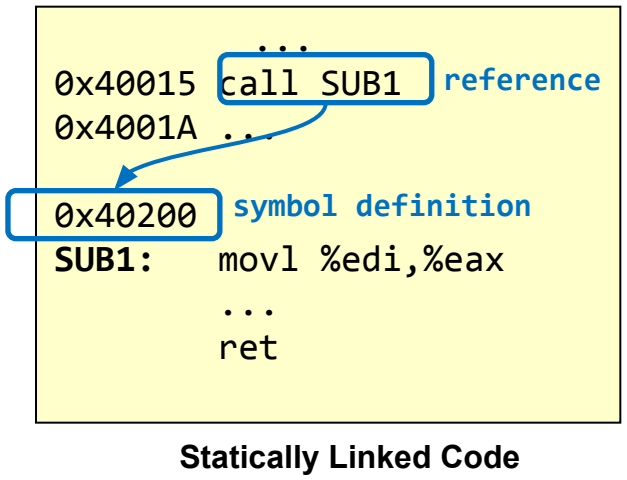
- Don't duplicate common code in physical memory
- Allow libraries to be updated (recompiled) without needing to recompile the programs that use the libraries
- Key Idea:
  - Don't hardcode the addresses of functions or data
  - Instead, use a **level of indirection** (a lookup table)
    - Lookup the address of the data or code at run-time and then use whatever address is found

Shared Library Motivation

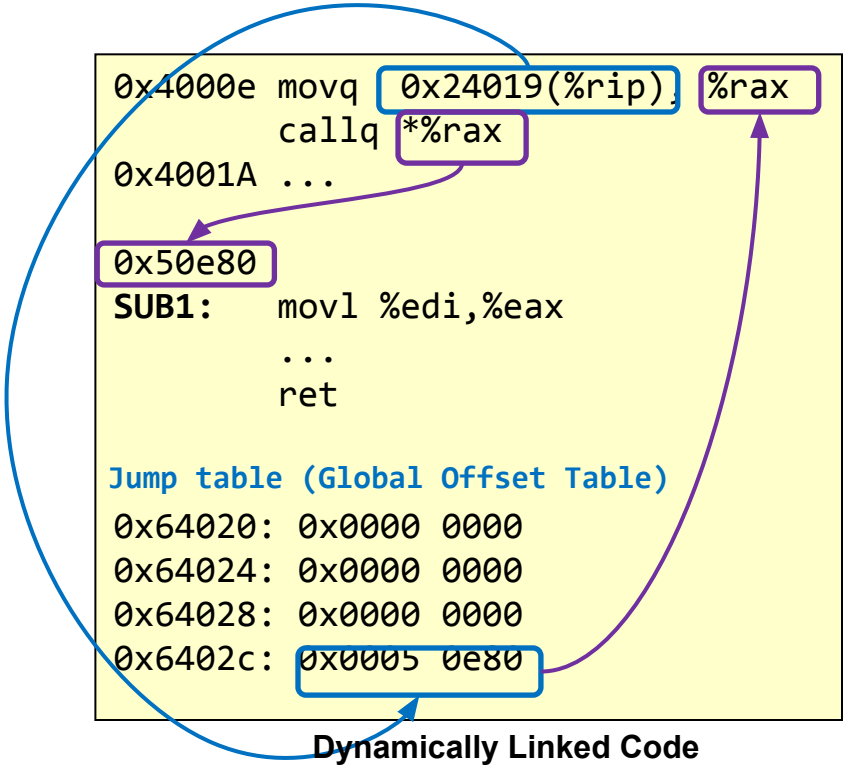


Simplified View of Dynamic Linkage

- Code can reference a table (aka Global Offset Table or "GOT") in the data section that will contain the address of the desired function or global variable
- Relocation entries in the executable object file will be used by the loader and dynamic linker **when the program is loaded** to fill in the table with the correct address
- More details in CS:APP3e



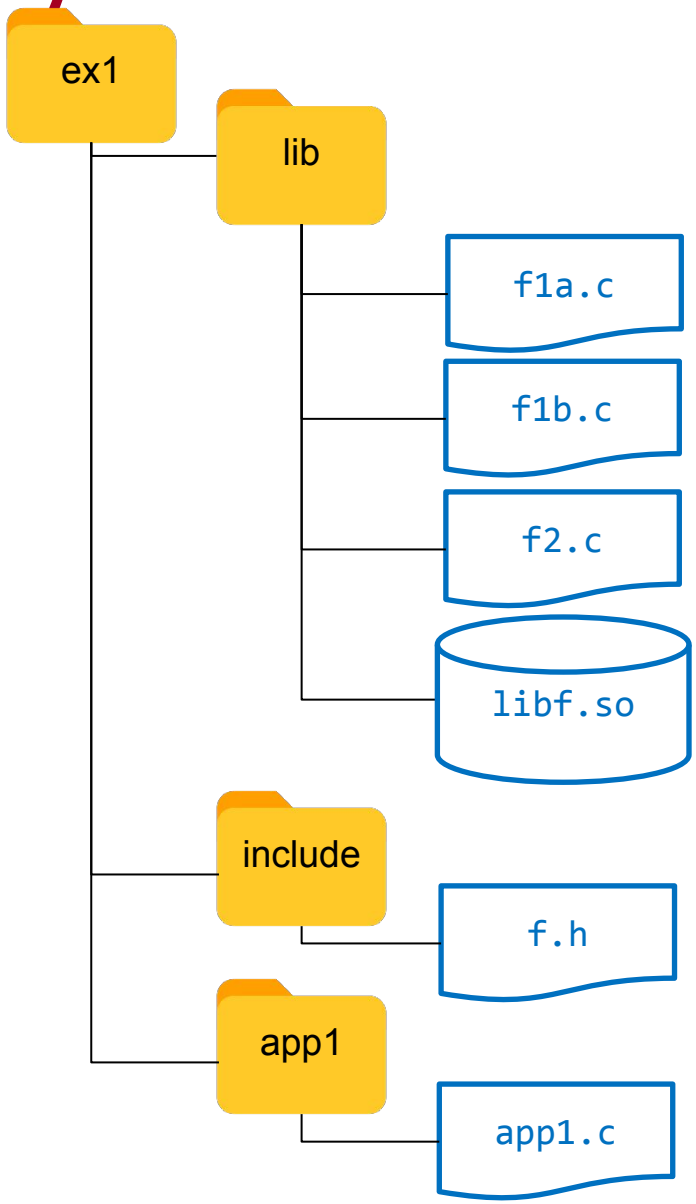
Statically Linked Code



Dynamically Linked Code

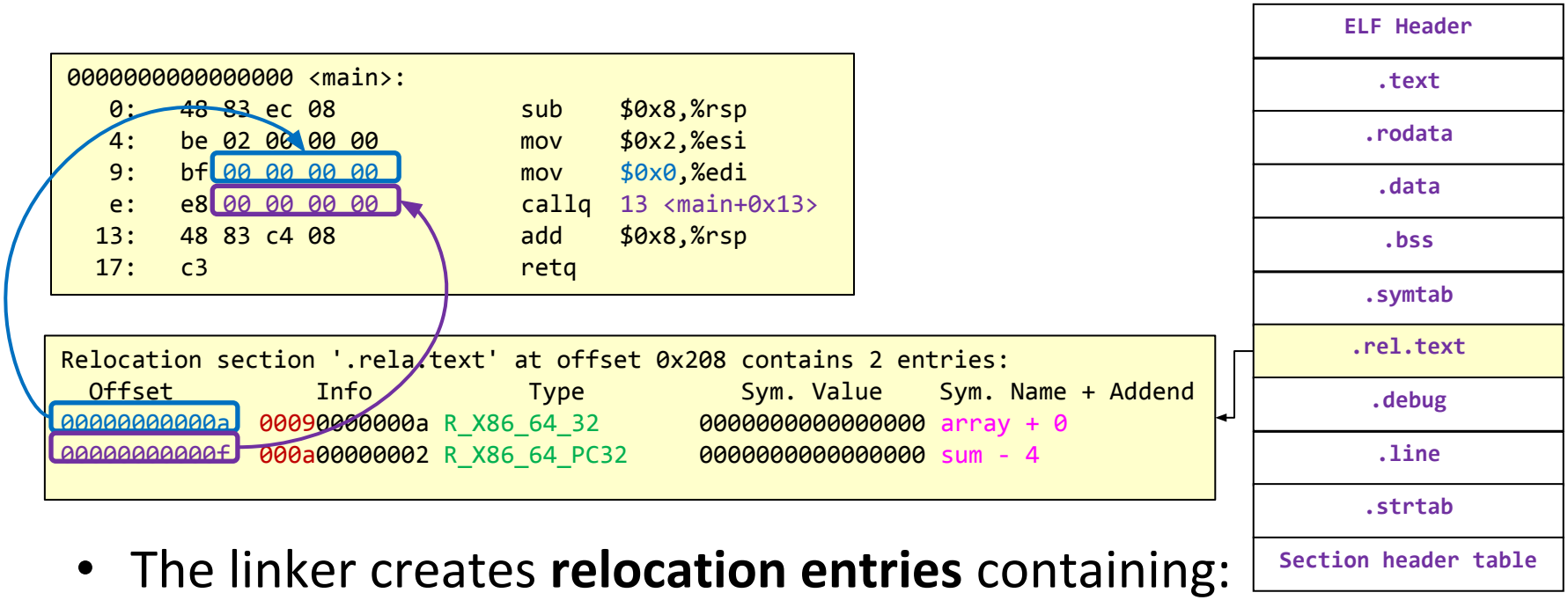
Try it! Shared Library Exercise

```
cd lib
rm -f *.o *.a
gcc -c -fpic f1a.c f2.c
gcc -shared f1a.o f2.o -o libf.so
ls
cd ../app1
gcc -I../include -L../lib app1.c -lf
./a.out       # loader can't find libf.so
# set search path for libraries
export LD_LIBRARY_PATH=../lib:$LD_LIBRARY_PATH
./a.out       # should see '11' output
cd ../lib
rm libf.so
gcc -c -fpic f1b.c
gcc -I../include -shared f1b.o f2.o -o libf.so
cd ../app1
./a.out       # should se '1111' output
              # without recompile/relink
cd ..
```



APPENDIX – DETAILS OF CALCULATING RELOCATIONS

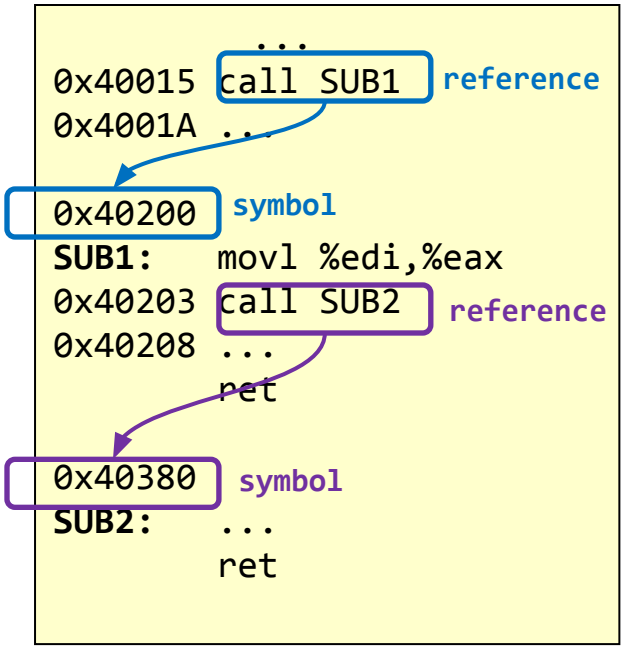
Relocation Entries



- The linker creates **relocation entries** containing:
  - Type of offset
    - **PC-Relative** (R_X86_64_PC32) or **Absolute Address** (R_X86_64_32)
  - Section offset where the relocation should be replaced
  - Symbol being referenced (and its symbol table index)

Relative Displacement Calculation

- When the linker combines all text and data into their respective sections, it will be able to determine the address of the symbol
- Then it will apply the relocation entries and calculate the PC-relative displacements
- Formula
  - Disp = Symbol address - (reference address + addend)
  - Recall: the PC has moved on to the next instruction by the time the instruction executes
  - addend is usually some constant (often 4) to account for the fact that the PC no longer points at the reference location
- Updated formula
  - Disp = Symbol address - (reference address + 4)
  - Disp = Symbol address + (-4) - reference address

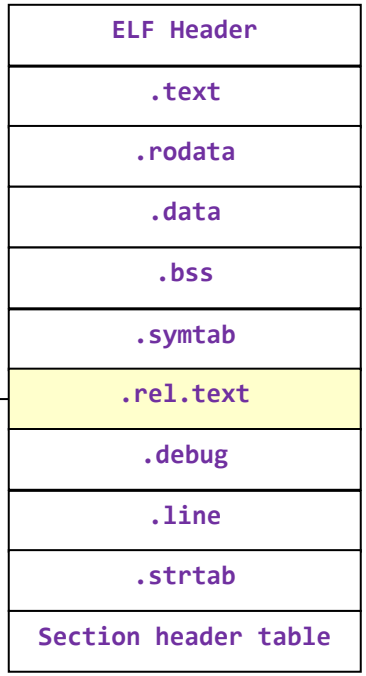


Applying Relocation Entries (1)

```
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
   e:   e8 00 00 00 00          callq  13 <main+0x13>
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

```
$ readelf -r main.o
```

```
Relocation section '.rela.text' at offset 0x208 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000a  00090000000a R_X86_64_32       0000000000000000 array + 0
00000000000f  000a00000002 R_X86_64_PC32     0000000000000000 sum - 4
```



Disp. = Symbol address + addend - reference address

0xffffffbf = 0x4004d6 + (-4) - 0x400513

Executable Object

```
00000000004004d6 <sum>:
  4004d6:       85 f6                   test   %esi,%esi
  ...

0000000000400504 <main>:
  400504:       48 83 ec 08             sub    $0x8,%rsp
  400508:       be 02 00 00 00          mov    $0x2,%esi
  40050d:       bf 30 10 60 00          mov    $0x601030,%edi
  400512:       e8 bf ff ff ff          callq  4004d6 <sum>
  400517:       48 83 c4 08             add    $0x8,%rsp
  40051b:       c3                      retq
```

Absolute Address Relocation:
***(main + 0x0a) = 0x601030**

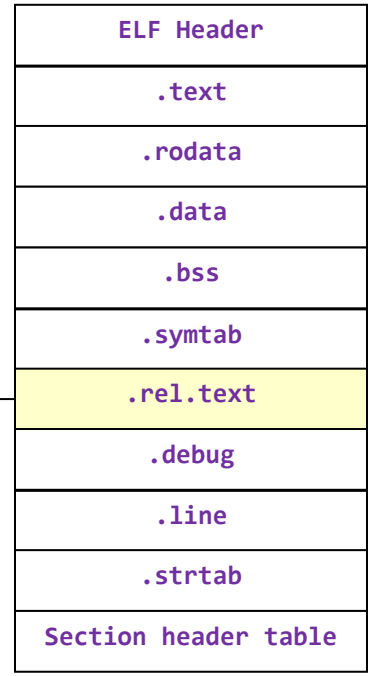
PC-Relative Relocation:
***(main + 0x0f) = 0xffffffbf =**
0x4004d6 + (-4) - 0x400513

Applying Relocation Entries (2)

```
0000000000000000 <sum>:
   0:   85 f6                   test   %esi,%esi
...
  26:   c6 05 00 00 00 00 01    movb   $0x1,0x0(%rip)
  2d:   c3                      retq
```

```
$ readelf -r sum.o
```

```
Relocation section '.rela.text' at offset 0x1d8 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000028  000900000002 R_X86_64_PC32     0000000000000001 done - 5
```



Disp. = Symbol address + addend - reference address

0x200b36 = 0x601039 + (-5) - 0x4004fe

Executable Object

```
0000000000601039 g     O .bss    0000000000000001              done

00000000004004d6 <sum>:
  4004d6:       85 f6                   test   %esi,%esi
...
  4004fc:       c6 05 36 0b 20 00 01    movb   $0x1,0x200b36(%rip) # 601039 <done>
  400503:       c3                      retq
```

PC-Relative Relocation:
***(sum + 0x28) = 0x00200b36**