Linking

B&O Readings: 7

CSE 361: Introduction to Systems Software

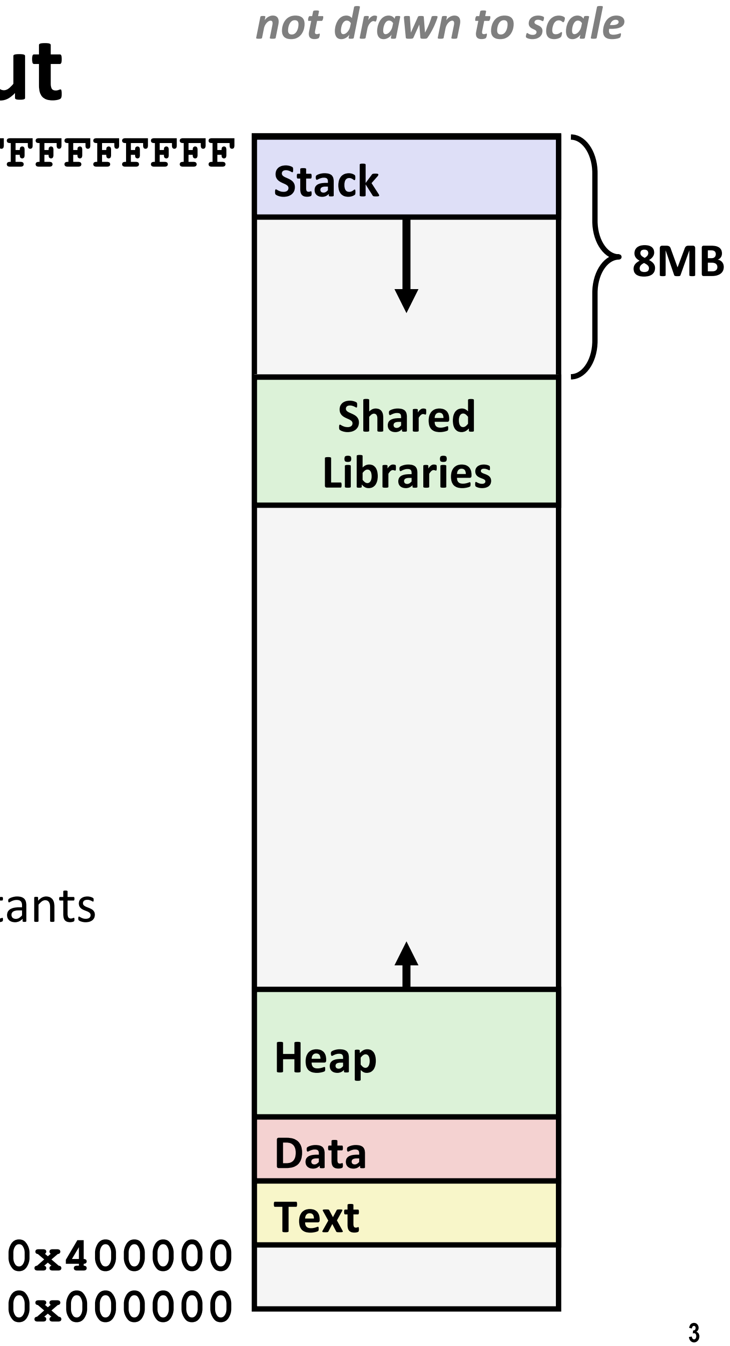
Instructor:
I-Ting Angelina Lee

Today

- **Linking**
  - What does a linker do
  - Motivation --- why linkers?
  - How it works
- **Static vs Dynamic libraries**
- **Case study: Library interpositioning**

x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
  - .bss, .data, .rodata
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only



Example C Program

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

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

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

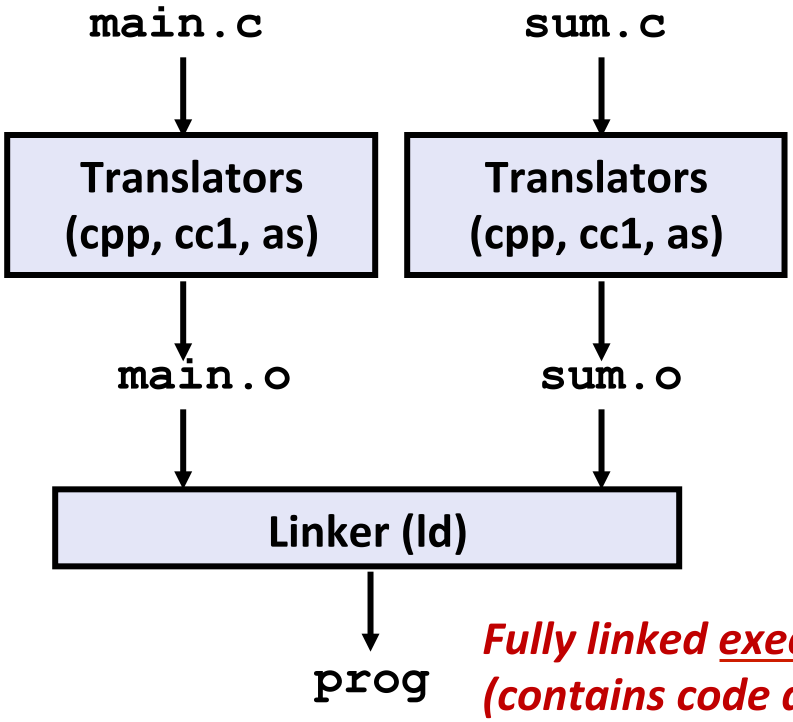
main.c

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

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

sum.c

Static Linking

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



Source files

Separately compiled
relocatable object files

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

Q: Where Does Each Variable Live?

```
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
```

- main: .text (function)
- array: .data (global variable)
- val: stack (local variable)
- argc, argv: stack (args)

- sum: .text (function)
- i, s: stack (local variables)
- a, n: stack (args)

- **When we compile main, how do we know where sum lives?**
- **Where is array located?**

Linker, at a Glance

```
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

- For every function / variable defined by your program that live in .text and .data, there is a symbol corresponds to it (naming the symbol = referring to its address).
- It's linker's job to figure out where each function / variable defined by your program goes and resolve references to these symbols.

Why Linkers?

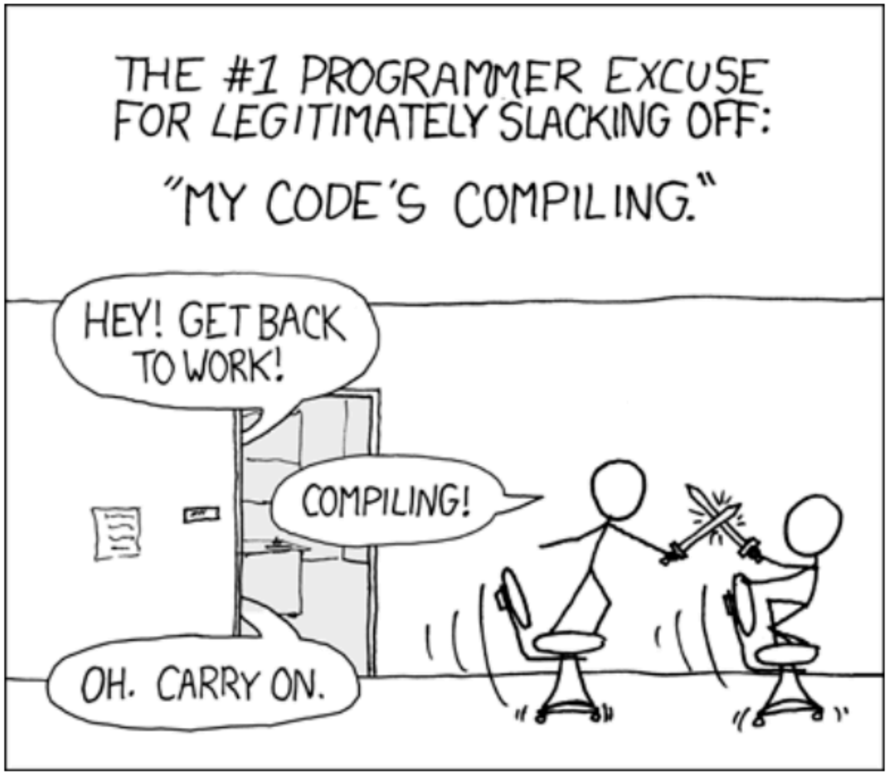
- **Reason 1: Modularity**

  - Program can be written as a collection of smaller source files, rather than one monolithic mass(/mess)

  - Can build libraries of common functions
    - e.g., Math library, standard C library

Why Linkers? (cont)

- **Reason 2: Efficiency**
  - **Time:** Separate compilation
    - Change one source file, compile, and then relink
    - No need to recompile other source files



- **Space:** Libraries
  - Common functions can be aggregated into a single file...
  - Executable files contain only code for the functions they actually use

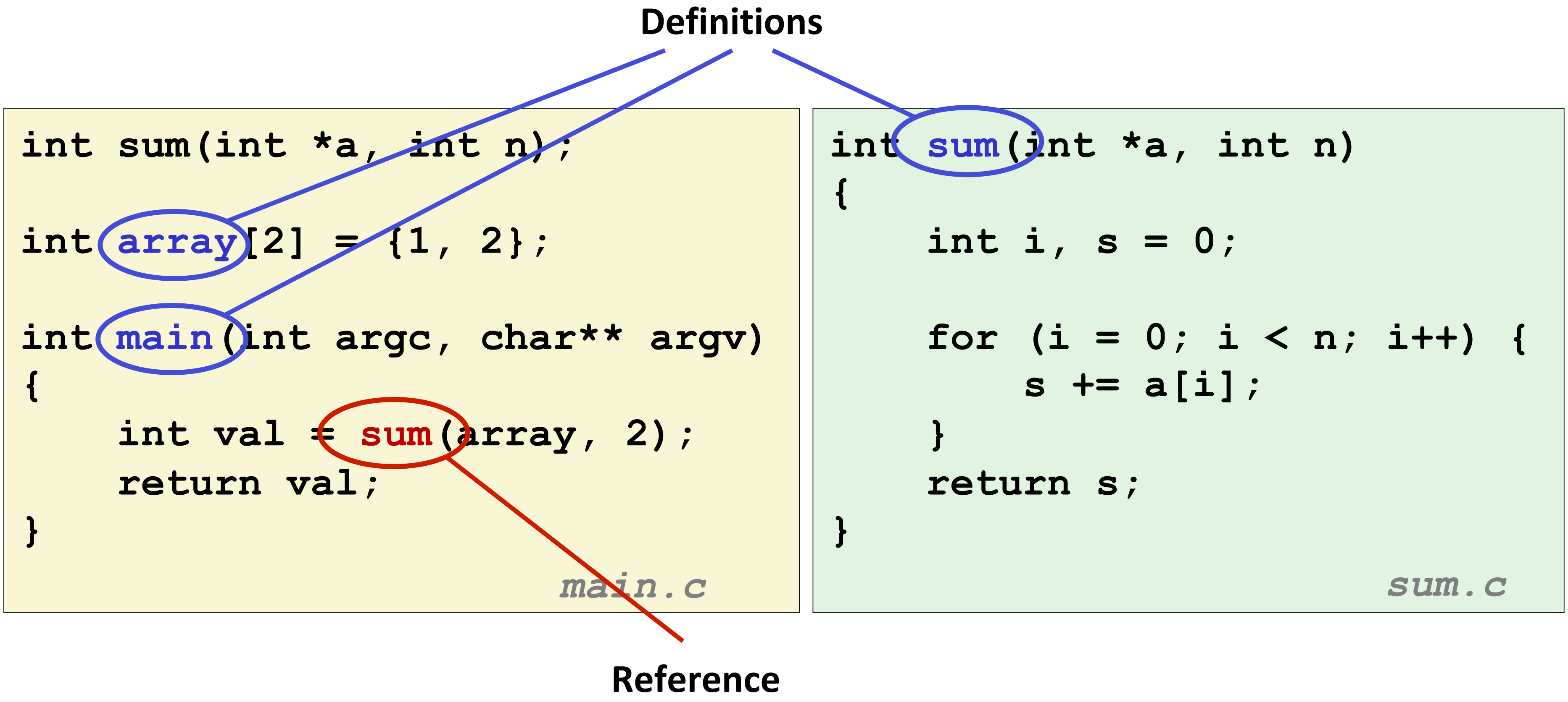
Why Should You Understand How Linking Works

- Understanding linkers help you build large programs.
- Understanding linkers help you decipher whether an "compiling error" comes from the compiler vs. the linker errors.
- Understanding linkers help you avoid difficult-to-debug programming errors.
- Understanding linkers help you understand how scoping rules are implemented.

What Do Linkers Do? (1)

- **Step 1. Symbol resolution**
  - Programs define and reference *symbols* (variables, 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***
    - An array of structs
    - Each entry: name, size, location of symbol
  - ***Linker associates each symbol reference with exactly one symbol definition***.

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

What Do Linkers Do? (2)

- **Step 2. Relocation**
  - **Merges** separate code and data sections into single sections (see picture on slide 15)
  - **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)

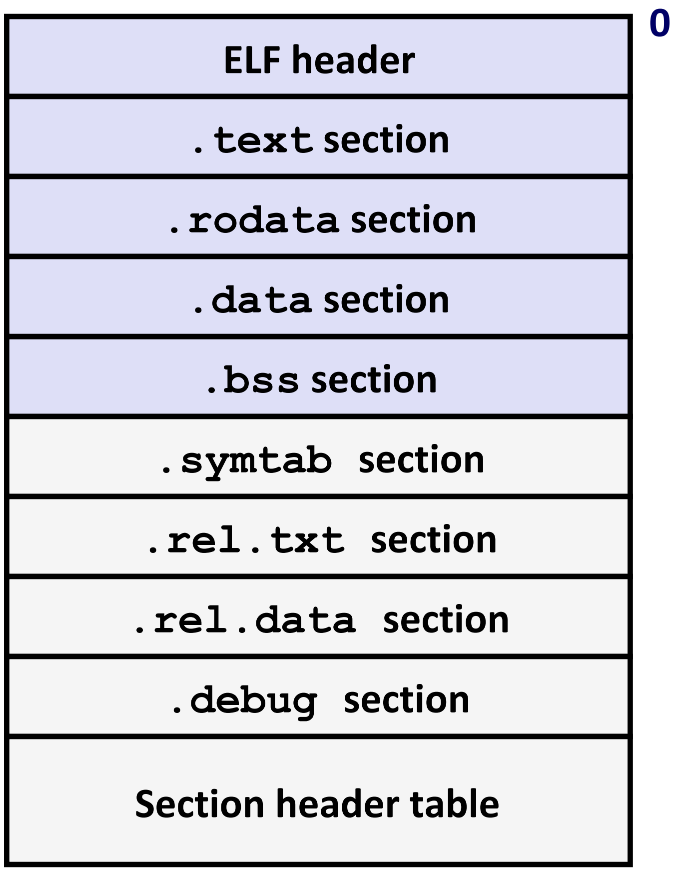
- **Standard binary format for object files**

- **One unified format for**
  - Relocatable object files (`.o`),
  - Executable object files `(a.out`)
  - 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.
- **`.text` section**
  - Code
- **`.rodata` section**
  - Read only data: jump tables, ...
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables or ones initialized with 0
  - "Block Started by Symbol" ("Better Save Space")
  - Has section header but occupies no space



0

ELF header

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.rel.txt` section

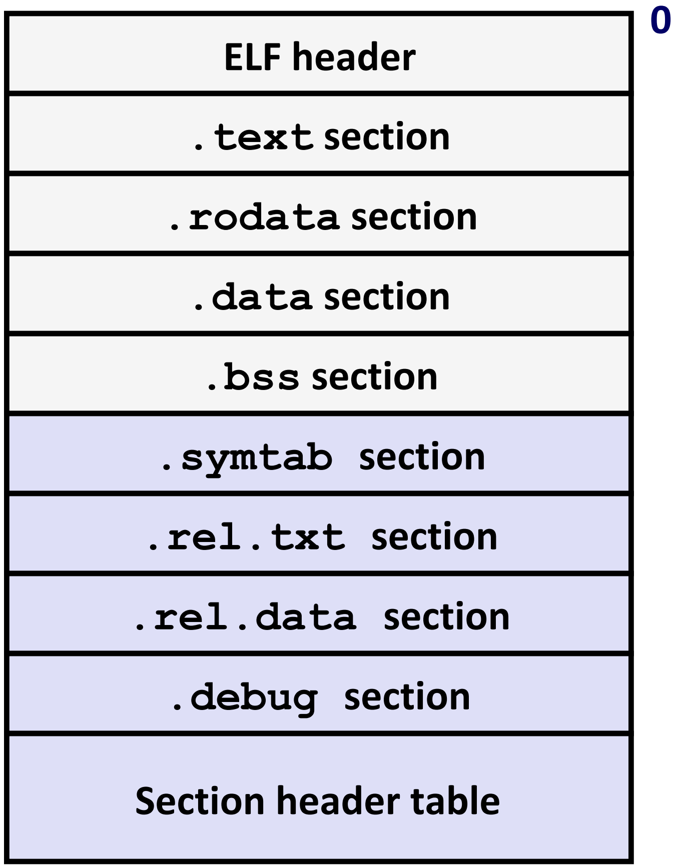
`.rel.data` section

`.debug` section

Section header table

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



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**

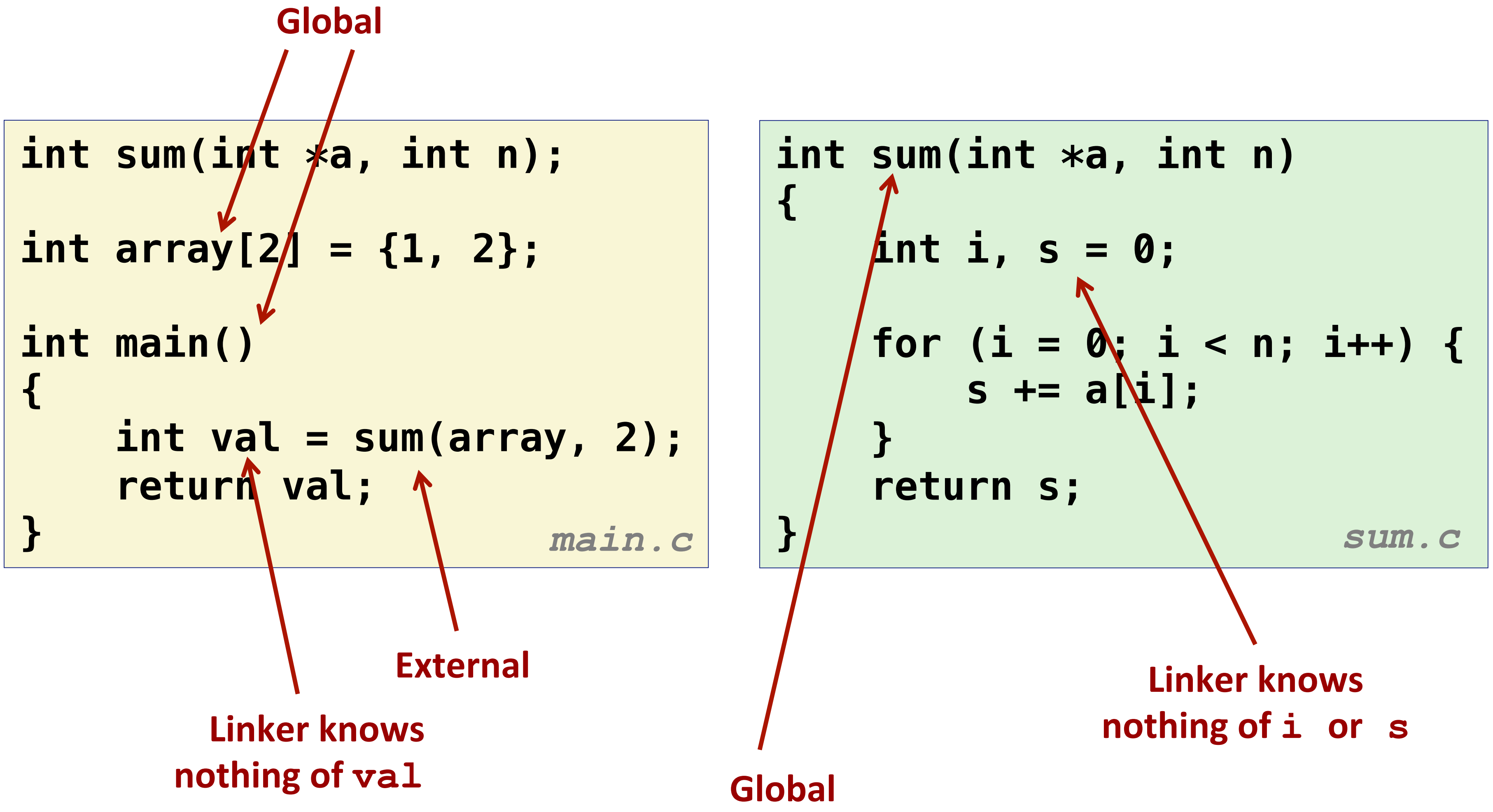
Digression: Static Variables

```
static int x = 99;

int foo()
{
    static int y = 0;
    y++;
    printf("%d\n", y);
}

int bar()
{
    x++;
    printf("%d\n", x);
}
```

- The non-local static variable (e.g., x) is only visible within this file. Reference to it in a different file will triggers an error from the linker.
- The increment of y takes effect each time `foo` is called!

What does the `printf` in `foo` print ?	1, 2, 3, …
What does the `printf` in `bar` print ?	100

Step 1: Symbol Resolution



Global

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

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

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

main.c

External

Linker knows nothing of `val`

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

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

sum.c

Global

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 time;

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

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

Names:

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

Local Symbols

```
static int x = 99;
int f()
{
     static int x = 0;
     return x++;
}

int g()
{
     static int x = 1;
     return x++;
}
```

The linker allocates space in `.data` for each definition of `x` and creates local symbols in the symbol table with unique names, e.g., `x.1`, x.2, x.3 … etc.

- **The local static variable shadows the non-local static variable with the same name.**
- **Local non-static C variables vs. local static C variables**
  - local non-static C variables: stored on the stack (linker knows nothing about these)
  - local static C variables: stored in either `.bss,` or `.data`

Resolving Symbols

Which symbols are global? Which are external? Which are local?

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

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

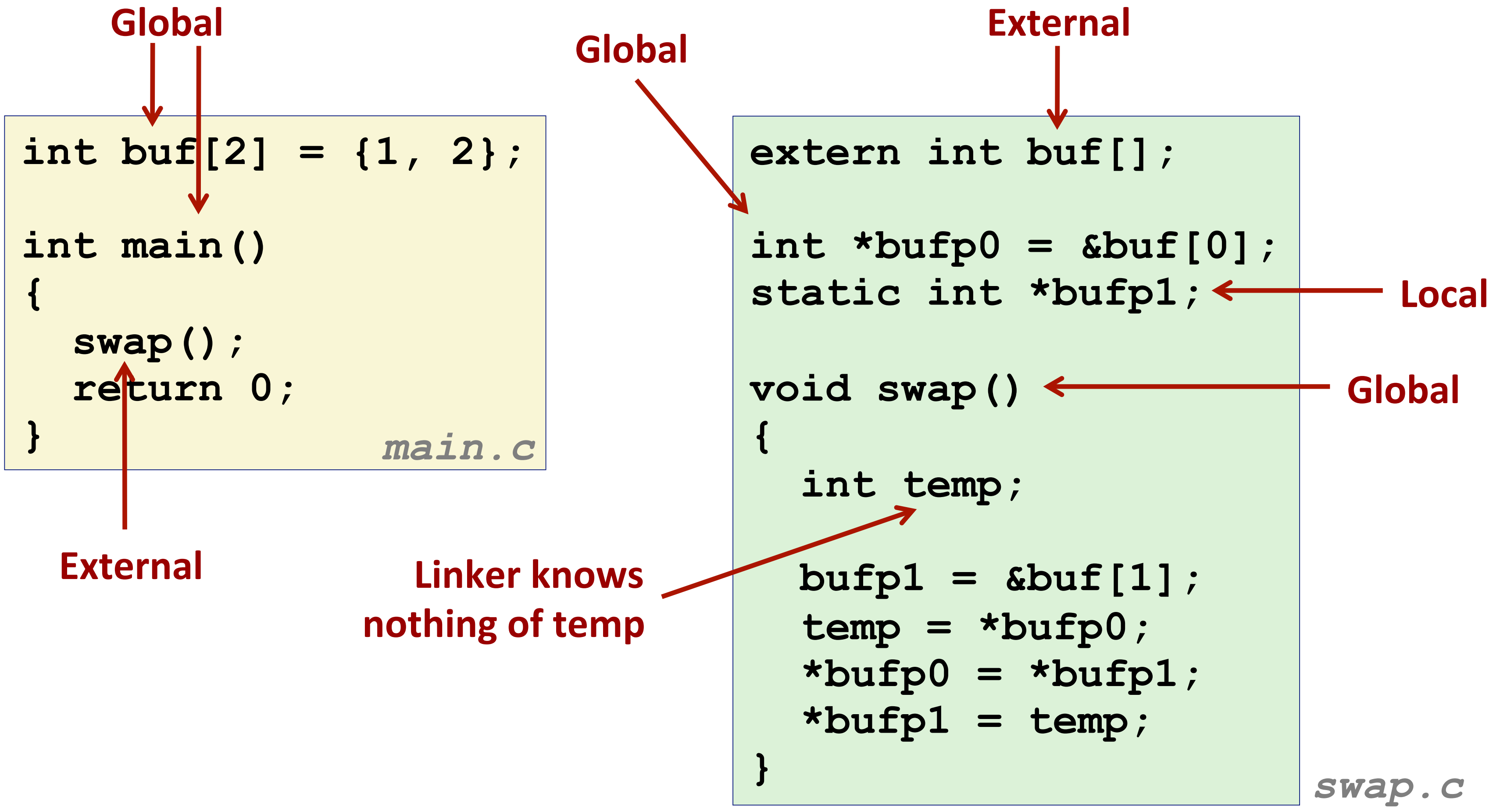
main.c

```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

Resolving Symbols

Which symbols are global? Which are external? Which are local?



Global

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

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

main.c

External

Global

External

```
extern int buf[];

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

void swap()
{
   int temp;

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

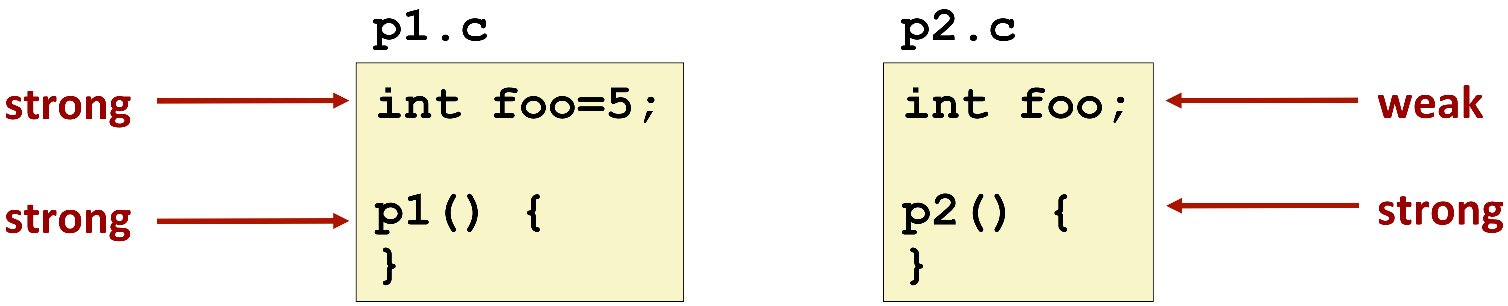
swap.c

Local

Global

Linker knows nothing of temp

How Linker Resolves Duplicate Symbol Definitions



`p1.c`

strong

```
int foo=5;

p1() {
}
```

strong

`p2.c`

weak

```
int foo;

p2() {
}
```

strong

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

Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error

- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol

- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc –fno-common`

Linker Puzzles

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

```

p1() {}
```

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

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

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

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

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

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

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

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

What happens when you compile each of these pair of files?

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.

What happens when you compile each of these pair of files?

Type Mismatch Example

```
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
- What gets printed?

Bit representation for double 3.14 interpreted as a long value.

Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use `static` if you can
  - Initialize if you define a global variable
  - Use `extern` if you reference an external global variable
    - Treated as weak symbol
    - But also causes linker error if no strong symbol defined in some file.
    - Unfortunately, you still don't get an error if the extern declares the wrong type!

Use of `extern` in .h Files (#1)

`c1.c`

```
#include "global.h"

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

`global.h`

```
extern int g;
int f();
```

`c2.c`

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

int g = 0;

int main(int argc, char argv[]) {
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Use of `extern` in .h Files (#1)

c1.c

```
~~#include "global.h"~~

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

```
extern int g;
int f();
```

c2.c

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

int g = 0;

int main(int argc, char argv[]) {
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

```
extern int g;
int f();
```

A compiler warning on mismatched type will trigger.

The compiler pre-processor simply dump dump the content of header file into the C file that includes it!

Use of .h Files (#2)

c1.c

```c
#include "global.h"

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

global.h

```c
#ifdef INITIALIZE
  int g = 23;
  static int local_init = 1;
#else
  extern int g;
  static int local_init = 0;
#endif
```

c2.c

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

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

Use of .h Files (#2)

c1.c

```
~~#include "global.h"~~

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

global.h

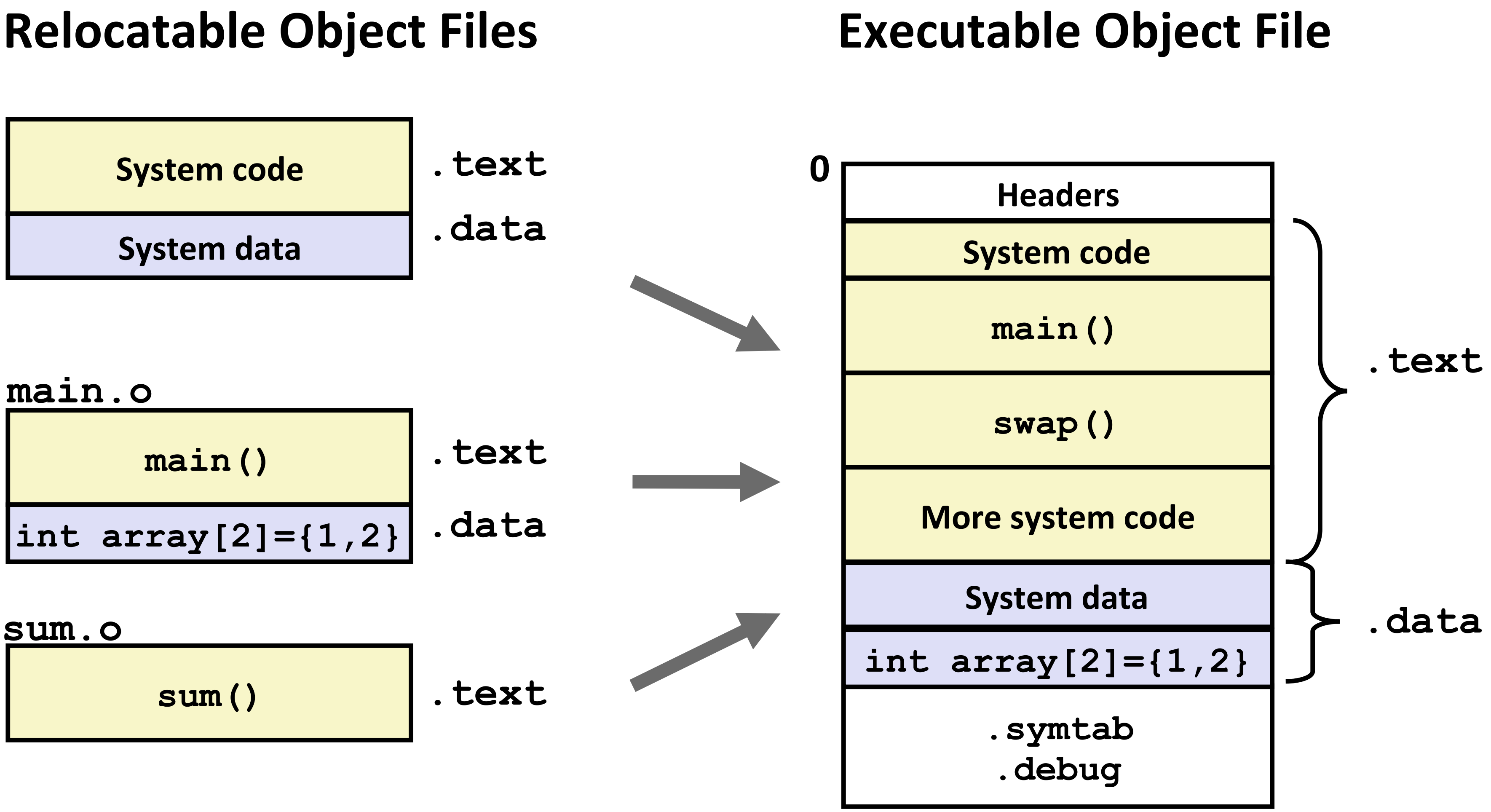
```
extern int g;
static int local_init = 0;
```

c2.c

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

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

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

Step 2: Relocation



Relocation Entries

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

int main()
{
    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 -rd 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
```

Using absolute addressing for array: 0x601018 (notice the little endian)

Source: `objdump -dx prog`

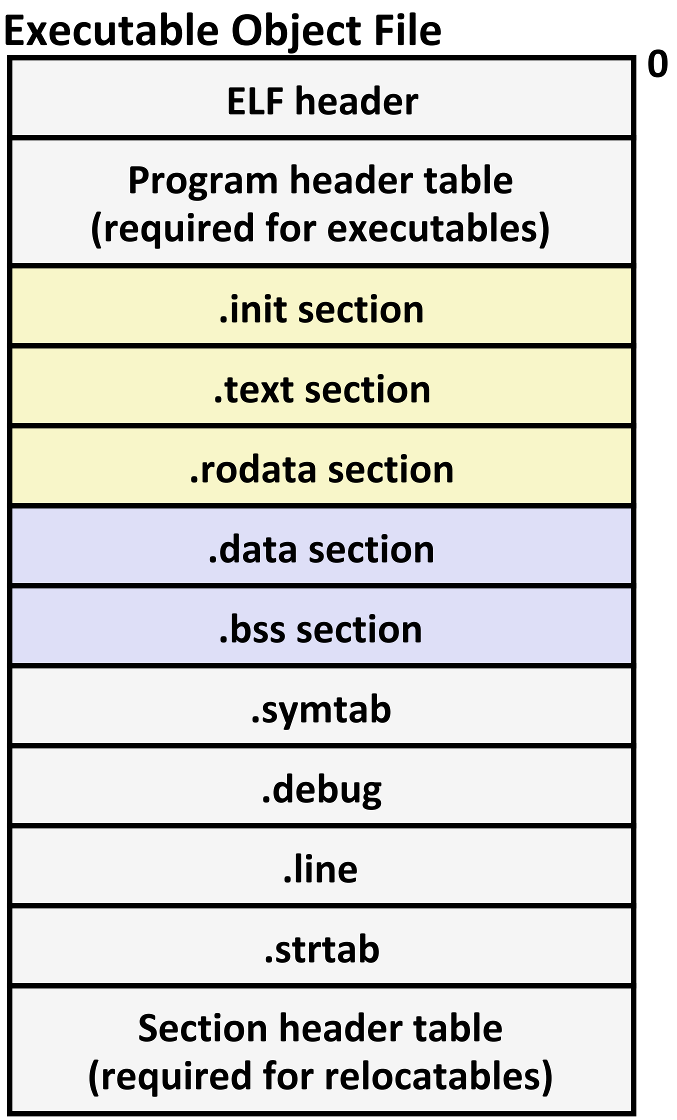
Relocated .text section

```
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
```

Using PC-relative addressing for sum(): 0x4004e8 = 0x4004e3 + 0x5

Source: `objdump -dx 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

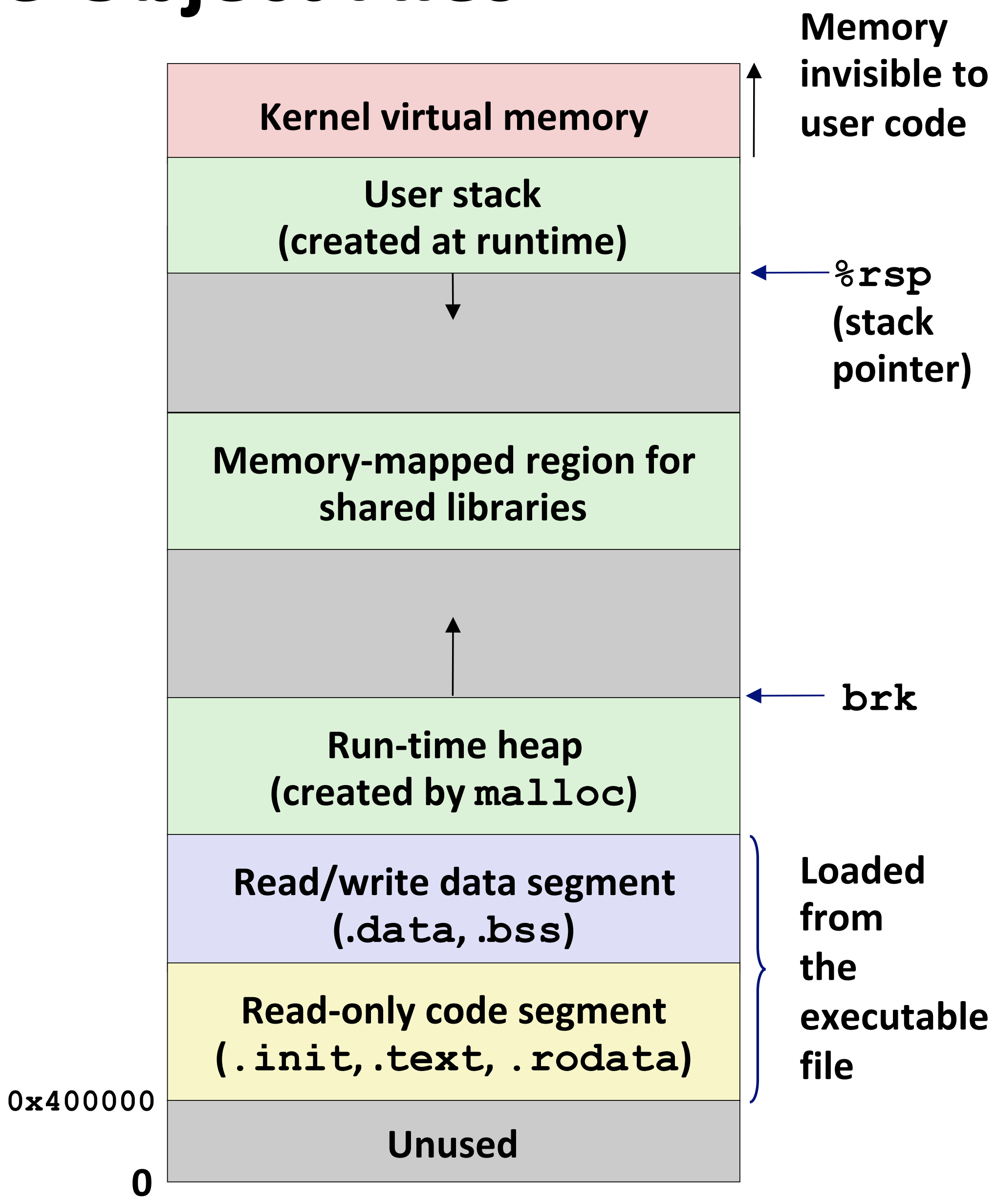
.symtab

.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

Recap: What We Learned Thus Far

- **The role of a linker**
  - symbol resolution and relocation
- **Scoping rules for global / static / extern variables**
- **What program entities have a linker symbol associated with it**
  - YES: functions, global variables, static variables
  - NO: local variables, function arguments
- **How linker resolves duplicate symbols**
  - strong vs weak symbols
  - the linker's symbol rules
- **How to use header files to avoid pitfalls in the use of global / static variables.**