CSE251: System Programming

9. Linking (1)

Seongil Wi

Department of Computer Science and Engineering

Recap: Executable and Linkable Format

A **standard file format for binary files (object files)** used on Unix and Unix-like systems

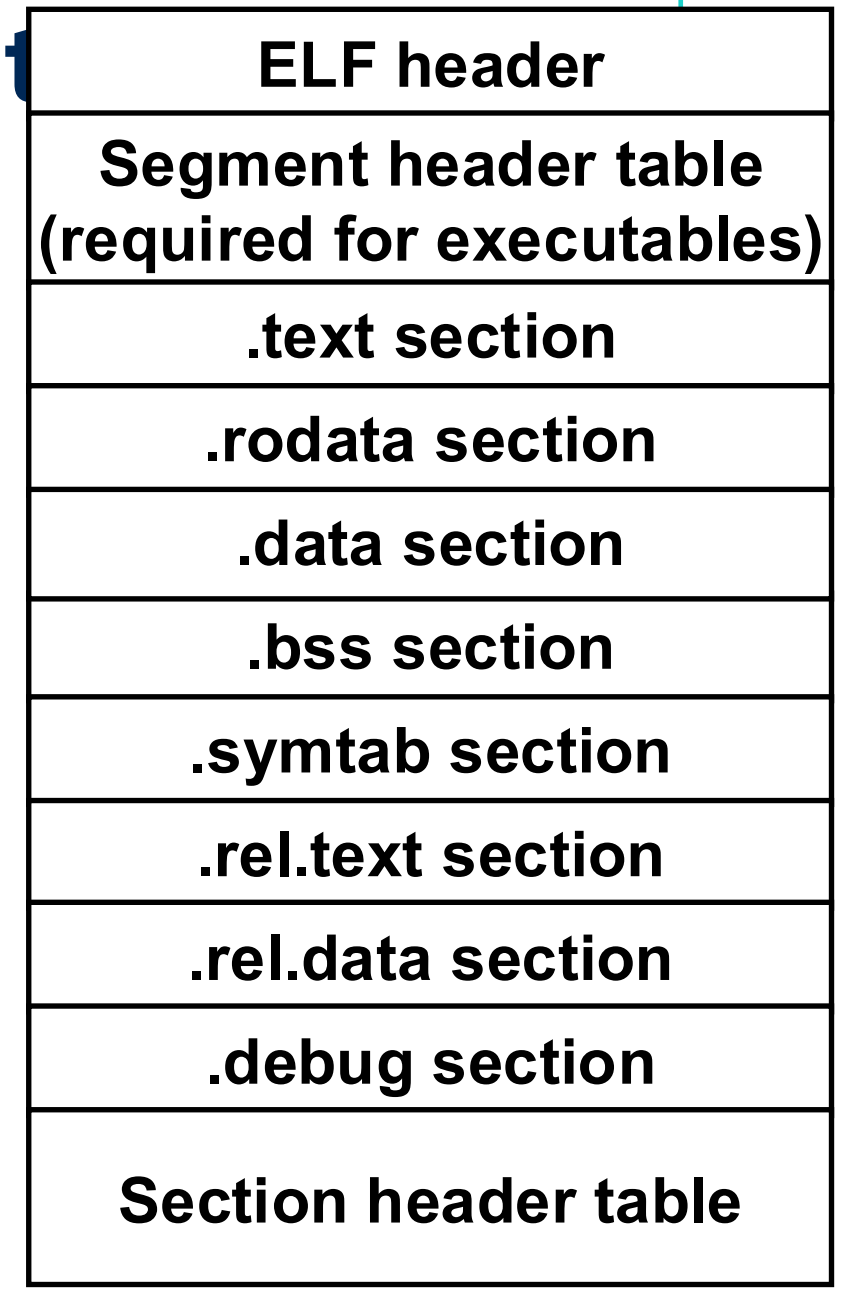
- C.f., Windows: Portable Executable (PE)
- C.f., macOS: Mach Object File Format (Mach-O)

- One unified format for
  - Relocatable object files (`.o`)
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

- Generic name: ELF binaries

Recap: ELF Object File Format

ELF has two views: the **section view** and the **segment view**:

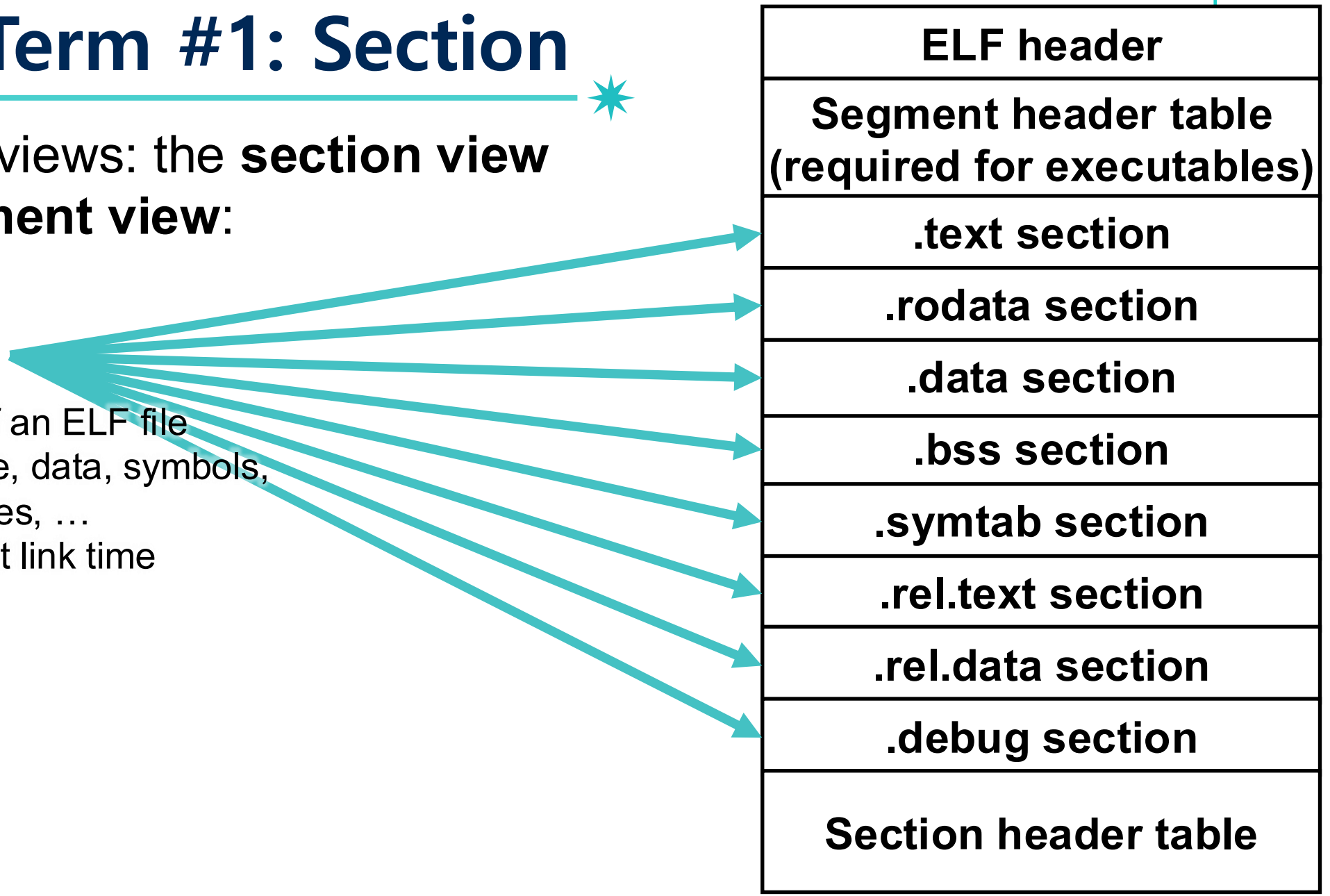


Recap: Term #1: Section

ELF has two views: the **section view** and the **segment view**:

Section

- A logical unit of an ELF file
- Organizes code, data, symbols, relocation entries, …
- Only relevant at link time



Recap: Term #2: Segment

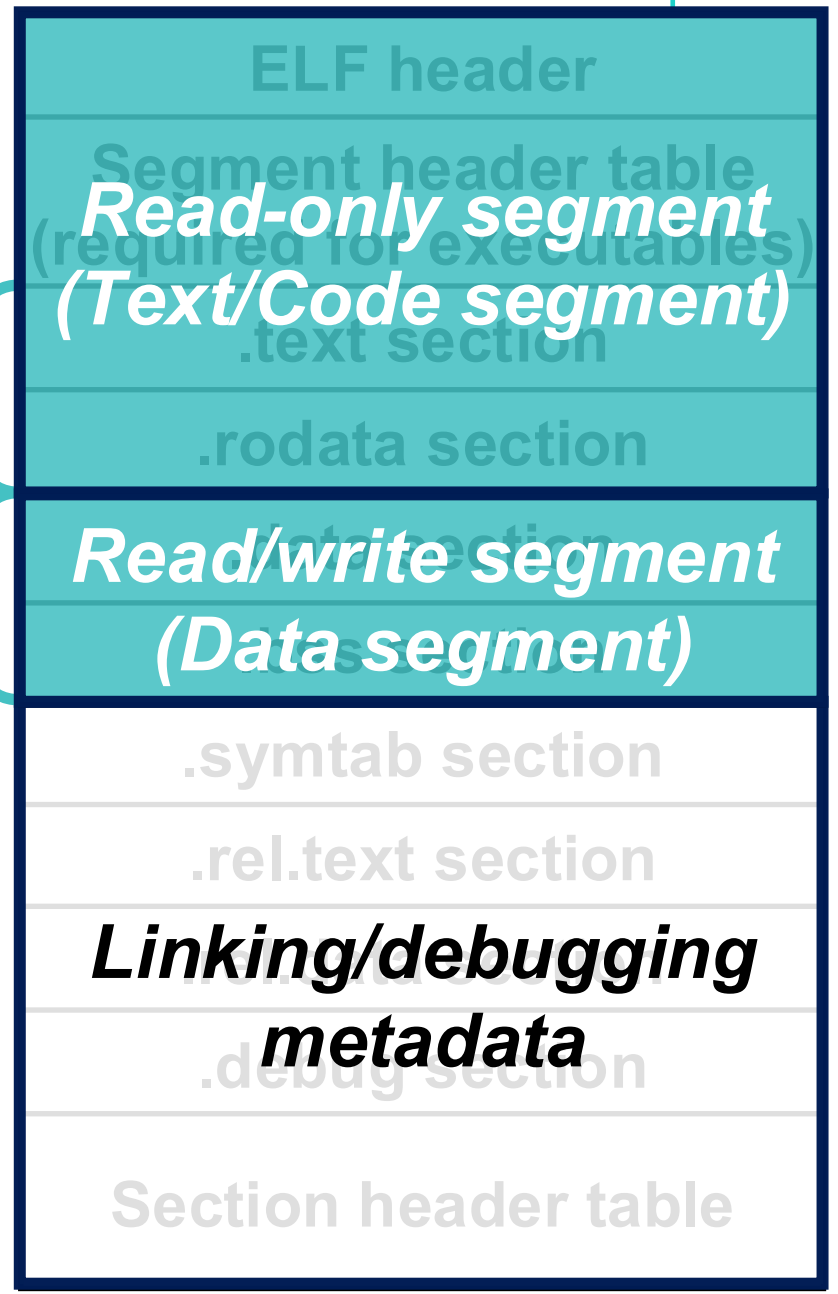
ELF has two views: the **section view** and the **segment view**:

Section

- A logical unit of an ELF file
- Organizes code, data, symbols, relocation entries, …
- Only relevant at link time

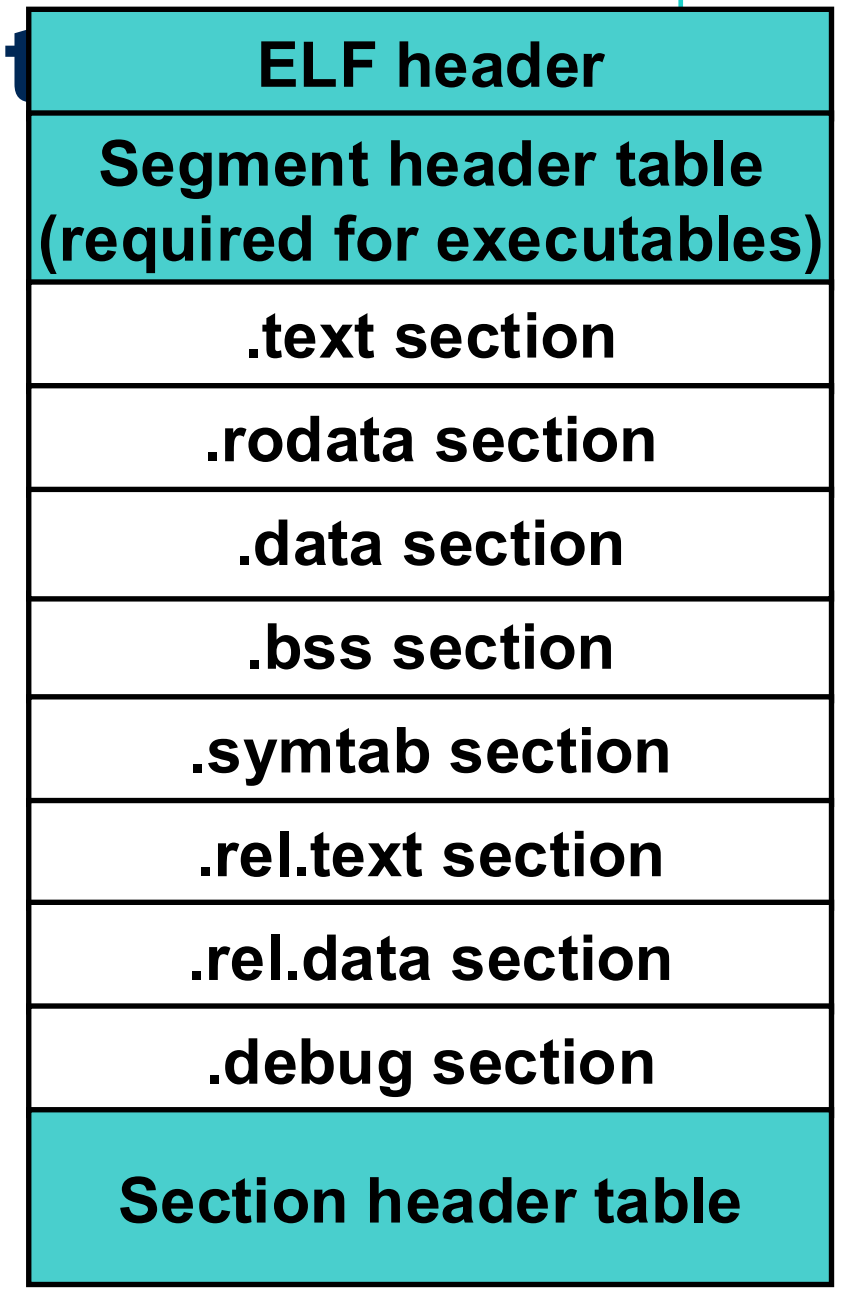
Segment

- Groups one or more sections into single segments
- Eventually be loaded into memory
- Only relevant at runtime



Recap: ELF Object File Format

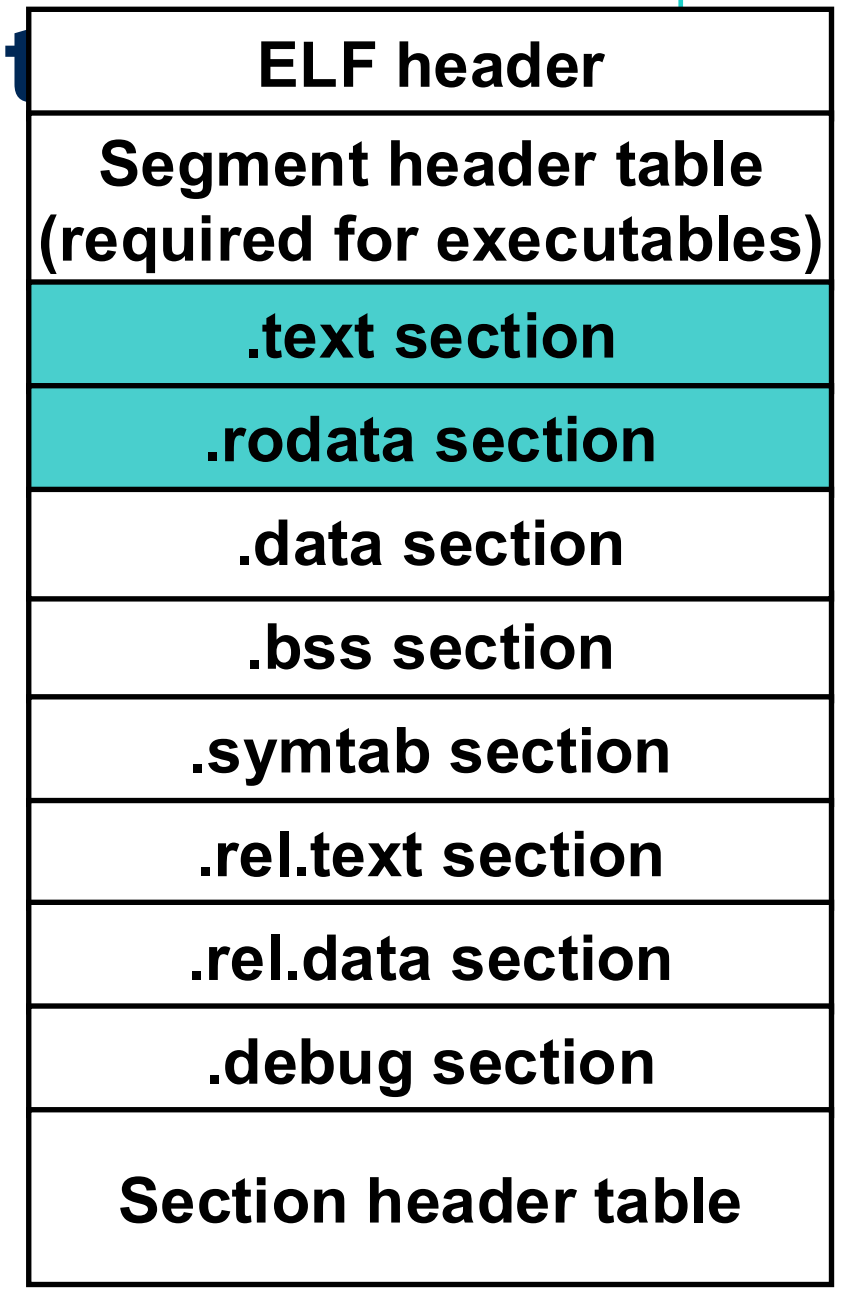
- **ELF header** (`$readelf -h <file>`)
  - Word size, byte ordering, file type (`.o`, `.so`, exec), machine type, etc.
- **Segment header table** (`$readelf -l <file>`)
  - Offsets and sizes of each segment
  - Page size, virtual address memory segments (sections), segment sizes (We will cover these in later.)
  - A.k.a., program header table
- **Section header table** (`$readelf -S <file>`)
  - Offsets and sizes of each section



Recap: ELF Object File Format

- **`.text` section**
  - Assembly code

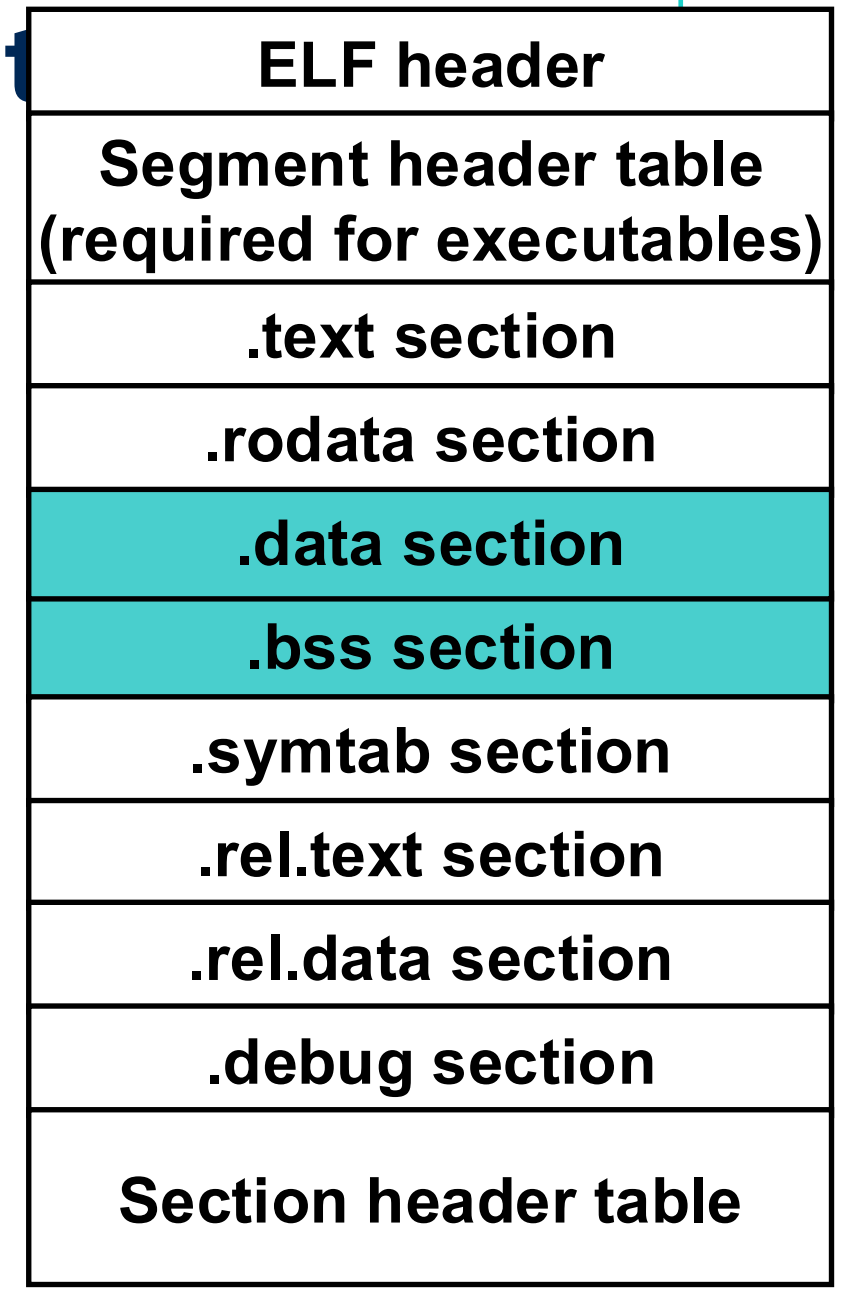
- **`.rodata` section** (`$readelf -p .rodata <file>`)
  - **r**ead-**o**nly **data**: string constants, jump tables, …
  - Values that are set once and don't change throughout the execution
  - E.g., `char *ptr = "Ube"; printf("Hello!");`



Recap: ELF Object File Format

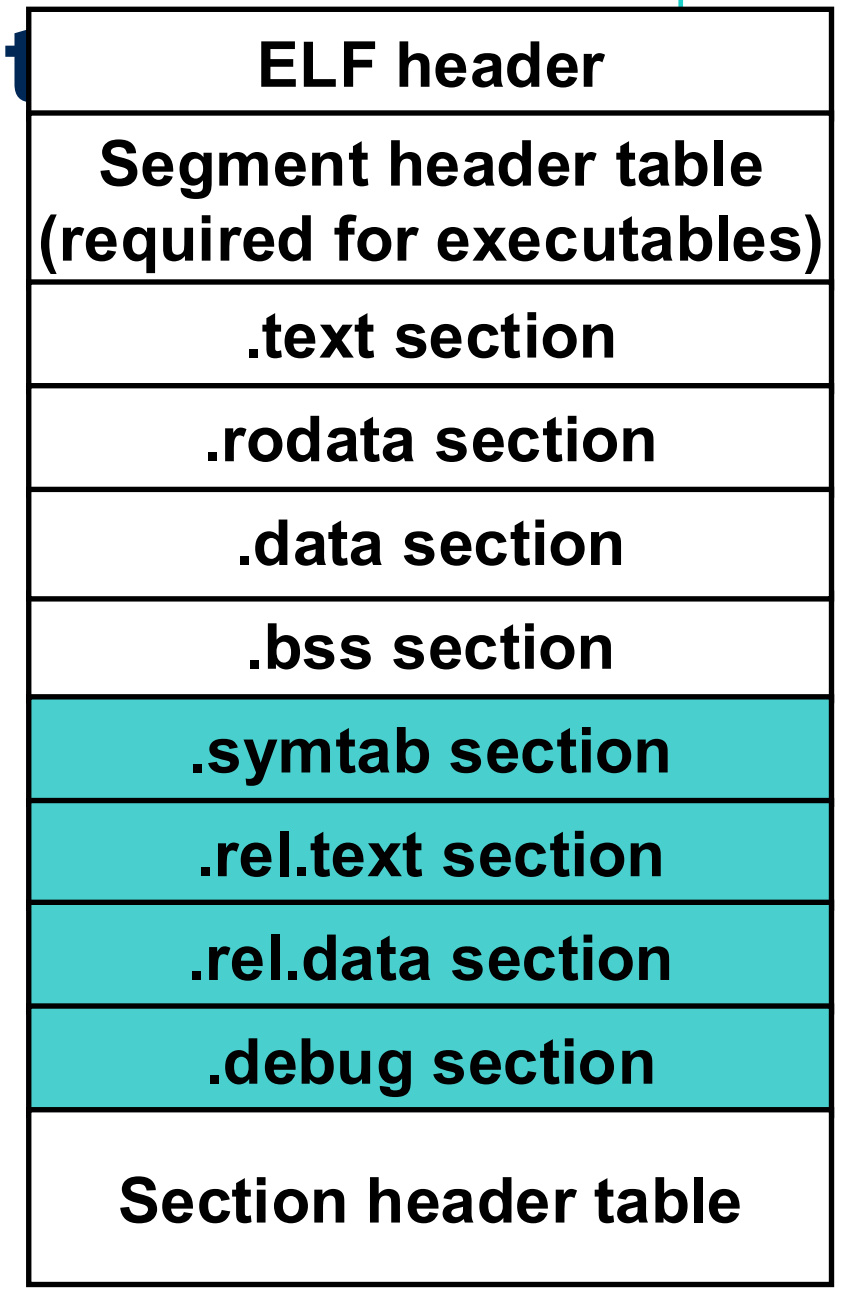
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables
  - Occupies no space in the executable file (save space)
  - When the program is loaded, memory for the `.bss` region is allocated and initialized to zero

```
int x = 1; // initialized global var
char a[20000000];     // uninitialized
global var

int main() {
    ...
}
```



Recap: ELF Object File Format

- **`.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`)

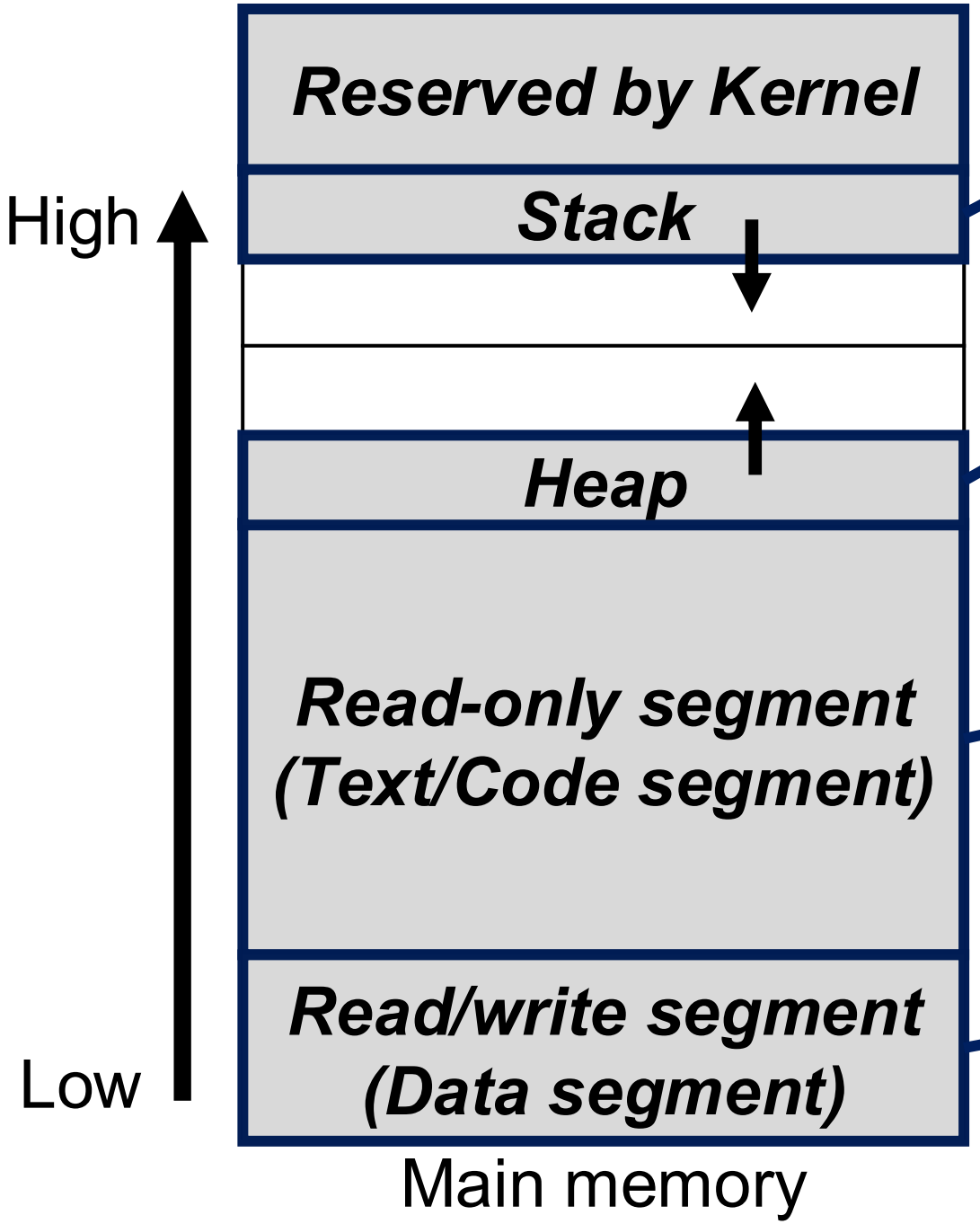


What Happens when You Run a Program?

1. The kernel reads the ELF executable and checks its headers
2. The kernel maps (loadable) segments into the program's memory address part
3. The kernel also sets up stack (`args`, env vars) and registers (`rip`, `rsp`, ...)
4. If the executable is dynamically linked, the kernel also loads the dynamic linker (`ld-linux.so`) specified in the ELF file
5. The kernel (or dynamic linker) transfers control the entry point of the binary: `_start`

We will cover about it later!

Who Invokes `main`?



- Stores local variables, function call information
- Dynamic memory for programmer to allocate (e.g., via `malloc()` or `realloc()`)
- Stores instructions being executed
- Stores (initialized and uninitialized) global variables

(Recap) Three Kinds of Object Files

- **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

Motivation

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

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

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

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

main.c

Problems?

Motivation

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

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

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

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

main.c

Problems?

It is tightly coupled to main.c, forming one monolithic block.

Motivation: Modularity!

```c
int sum (int *a, int n); // header

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

int main() {
    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

We refactor sum( ) into a library function for common reuse.

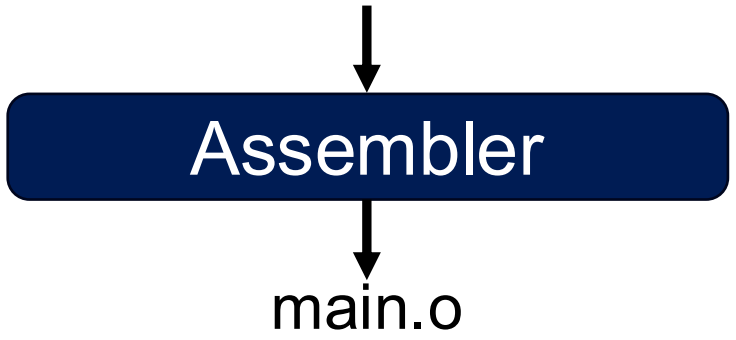
Motivation: Modularity!


```c
int sum (int *a, int n); // header

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

int main() {
    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



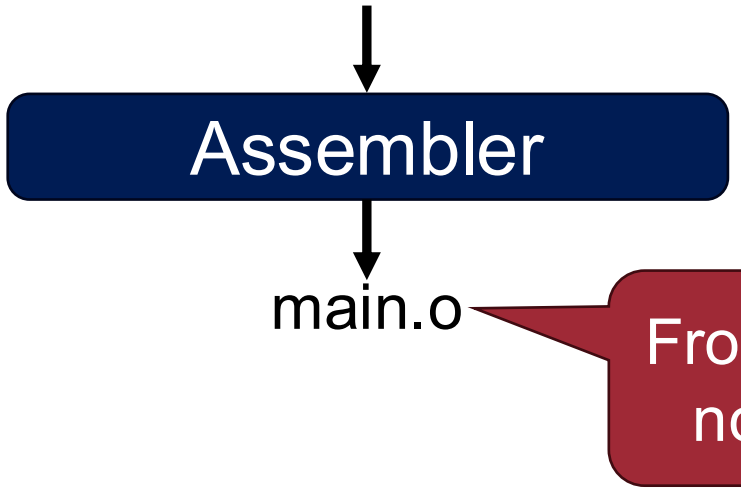
Motivation: Modularity!

```c
int sum (int *a, int n); // header

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

int main() {
    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



From main.o's perspective, it does not know where sum is defined

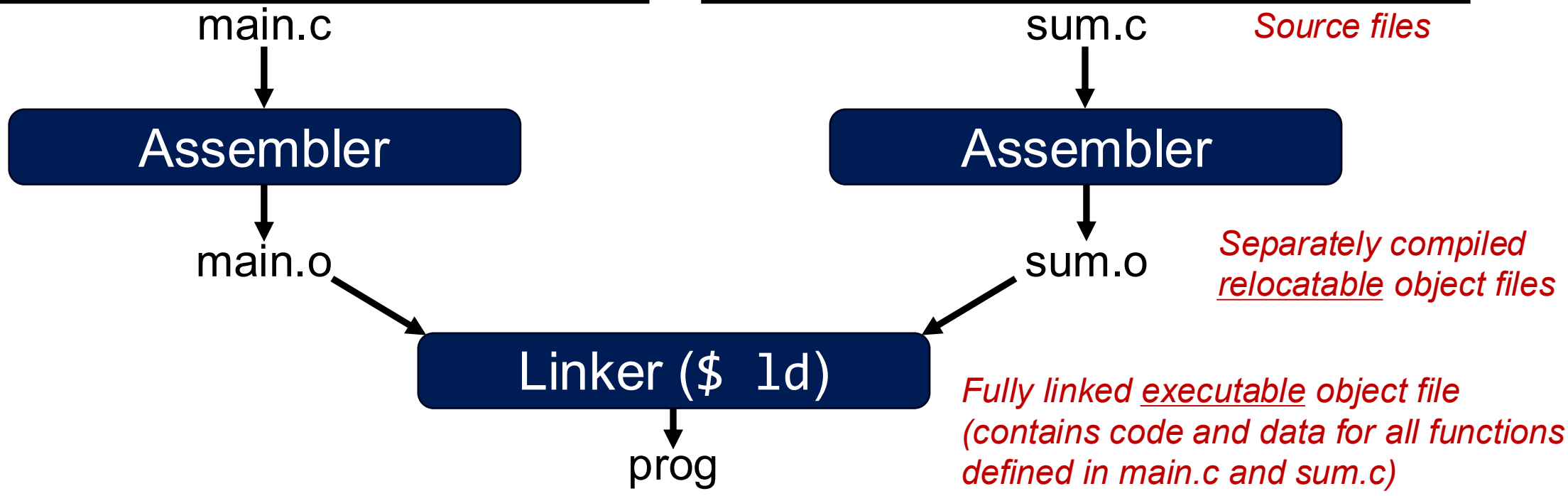
→ *We use linker!*

Linking

```
$ gcc -o prog main.c sum.c
$ ./prog
```

```c
int sum (int *a, int n); // header

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

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

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

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



Linking

Linker

- A computer program that combines object and library files **into a single executable file**

Why Linkers?

- **Reason 1: Modularity!**
  - Program can be written as a collection of smaller source files, rather than one monolithic mass
  - Can build libraries of common functions
    - e.g., Math library, standard C library (`libc`)
    - Header files in C (e.g., `stdio.h`) declare types that are defined in libraries

```c
int sum (int *a, int n); // header

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

int main() {
    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;
}
```

Common function

sum.c

Why Linkers?

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

If the implementation of this function changes, we only need to recompile this file and relink

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

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

int main() {
    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;
}
```

Common function

sum.c

Why Linkers?

- **Reason 2: Efficiency!**
  - Time: Separate compilation
    - Change one source file, compile, and then relink
    - No need to recompile other source files
    - Can compile multiple files concurrently
  - Space: Libraries
    - Common functions can be aggregated into a single file. Therefore, we need to manage space!
    - **Option 1: *Static Linking***
      - ✓ Executable files contain only the library functions they actually use
    - **Option 2: *Dynamic linking***
      - ✓ Executable files contain no library code
      - ✓ During execution, single copy of library code can be shared across all executing processes

Linking Process

Step 1: Symbol Resolution → **Step 2**: Relocation

Linking Process

Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

main.c

main.o

`sum` is not defined in `main.o`, so the linker must **resolve this symbol** by finding its definition elsewhere

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

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

sum.c

sum.o

Linking Process

Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

main.c

main.o

Symbol table:
- sum: undefined
- aray: defined
- main: defined

Resolution result:
- sum: defined
  in sum.o

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

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

sum.c

sum.o

Symbol table:
- sum: defined

Step 1: Symbol Resolution

- The linker associates each symbol reference with **exactly one symbol definition**

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

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

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

main.c

main.o

Symbol table:
- sum: undefined
- aray: defined
- main: defined

Resolution result:
- sum: defined in sum.o

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

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

sum.c

sum.o

Symbol table:
- sum: defined

What is the symbol (and symbol table)?

Symbols

symbols = global/static variables and functions

Symbols

symbols = global/static variables and functions

Programs **define** and **reference** symbols:

```
void swap() { … }
```
/* define symbol swap */ *(global function)*

```
swap();
```
/* reference symbol swap */ *(global function)*

```
int *xp = &x;
```
/* define symbol xp, *(global variable)*
reference symbol x */ *(global variable)*

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

(FYI) `static` Variables in C

- A variable whose **lifetime lasts for the entire execution** of the program

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

x is initialized only once

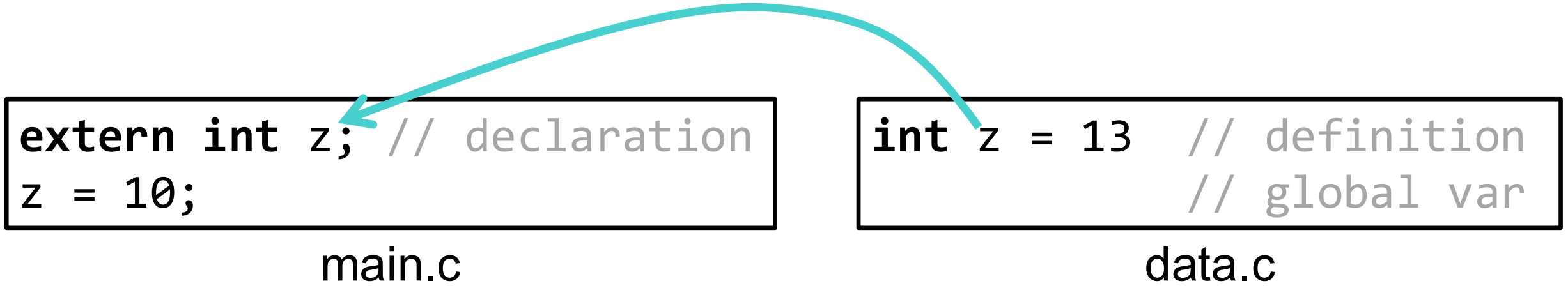
- First call: returns 1
- Second call: returns 2
- Third call: returns 3

x remains alive for the whole program execution

- **Inside a function**
  → the variable keeps its value across function calls
- **At global scope**
  → the variable is visible only within that source file

(FYI) extern Variables in C

- A variable that is declared in one file but defined elsewhere



Linker Symbols


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

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

main.c

```c
static int x = 1;
int z = 2;

int sum (int *a, int n) {
    static int y = 1;
    int i, s = 0;

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

sum.c

Linker Symbols

Global symbols: symbols defined by module *m* that can be referenced by other modules

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

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

main.c

```
static int x = 1;
int z = 2;

int sum (int *a, int n) {
    static int y = 1;
    int i, s = 0;

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

sum.c

Linker Symbols

Global symbols: symbols defined by module *m* that can be referenced by other modules

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

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

main.c

```
static int x = 1;
int z = 2;

int sum (int *a, int n) {
    static int y = 1;
    int i, s = 0;

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

sum.c

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

Linker Symbols (Final)

Global symbols: symbols defined by module *m* that can be referenced by other modules

Local symbols: symbols that are defined and referenced exclusively by module *m*

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

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

main.c

```
static int x = 1;
int z = 2;

int sum (int *a, int n) {
    static int y = 1;
    int i, s = 0;

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

sum.c

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

How About non-static Local Variables?

Linker knows nothing of local variables.
They are managed by stack!

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

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

main.c

```c
static int x = 1;
int z = 2;

int sum (int *a, int n) {
    static int y = 1;
    int i, s = 0;

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

sum.c

Recap: Linker Symbols

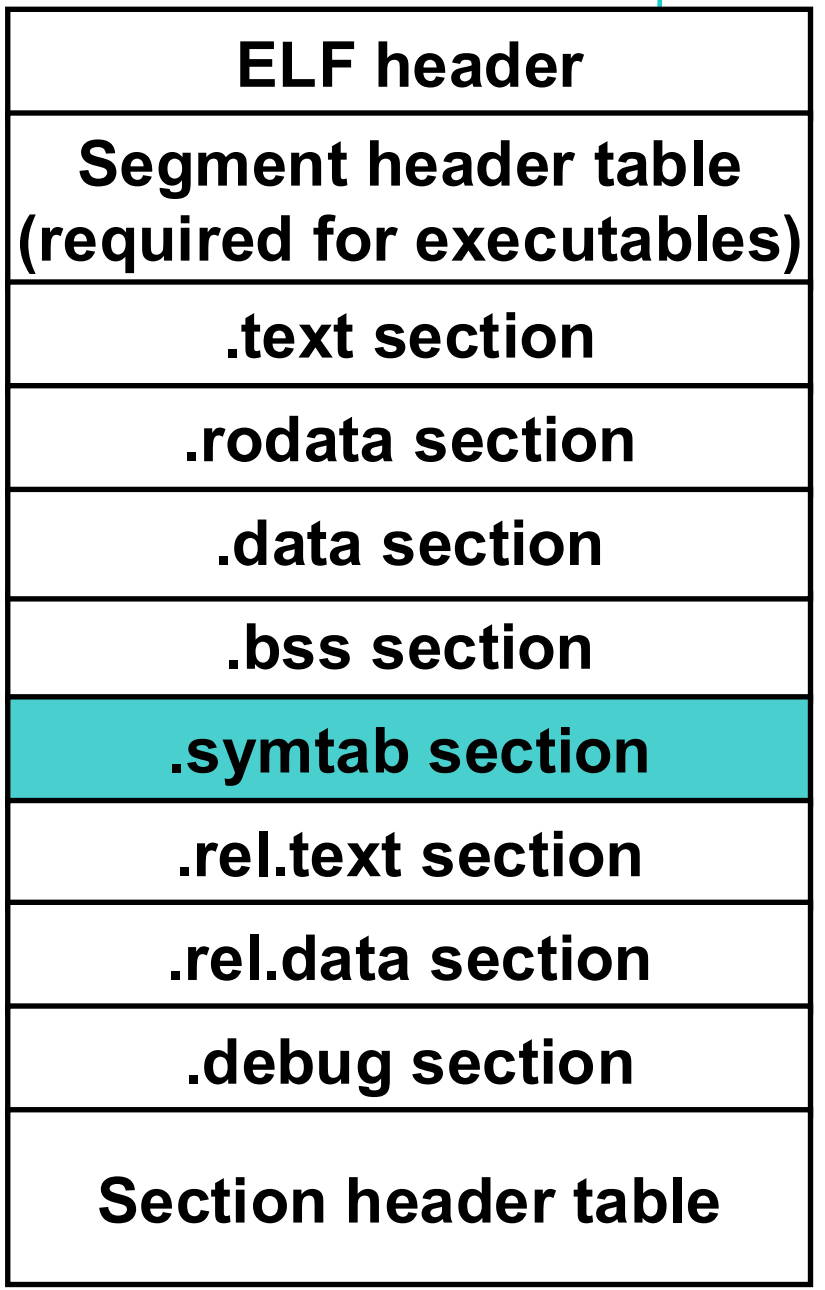
- **Global symbols**
  - Symbols defined by module *m* that can be referenced by other modules
  - e.g., non-`static` C functions and non-`static` global variables
- **External symbols**
  - Global symbols that are referenced by module *m* but defined by some other module
- **Local symbols**
  - Symbols 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

Symbol Table ($ readelf -s <objectfile>)

- Symbols are stored in object file (by assembler) in symbol table
  - Symbol table is an array of entries
  - Each entry includes name, size, and location of symbol

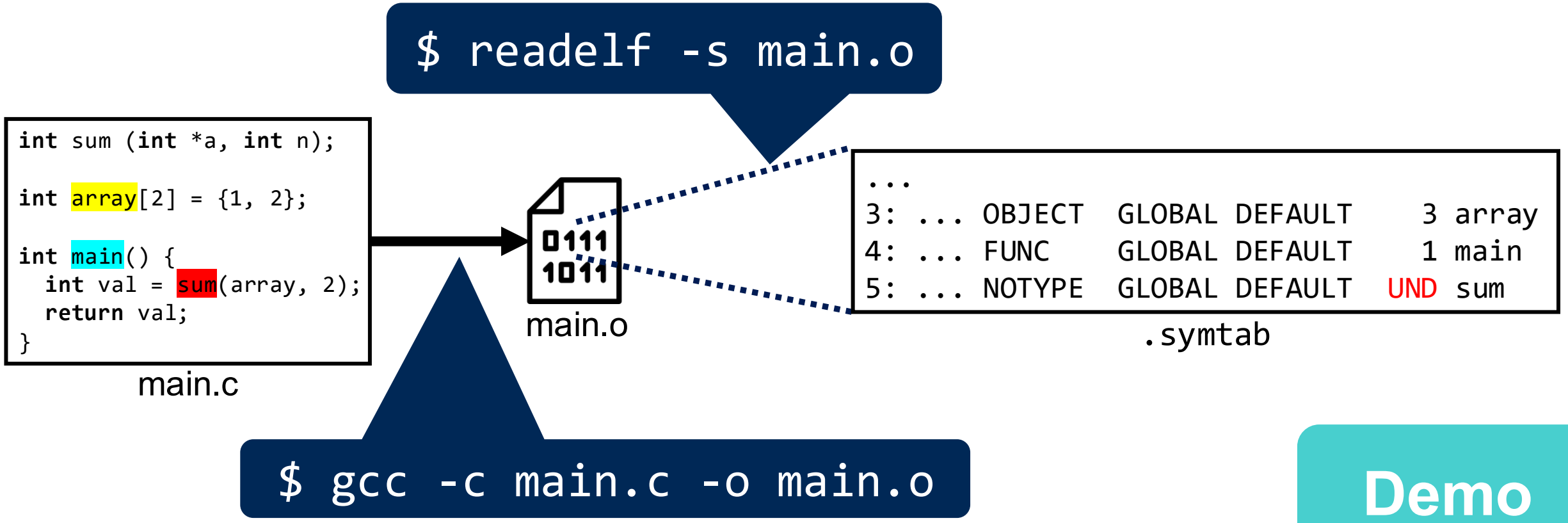
ELF Object File Format

- **`.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`)



Symbol Table ($ readelf -s <objectfile>)

- Symbols are stored in object file (by assembler) in symbol table
  - Symbol table is an array of entries
  - Each entry includes name, size, and location of symbol



Demo

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;
}
```

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`

The compiler gives unique symbol names because they are distinct objects

Step 1: Symbol Resolution

- The linker associates each symbol reference with **exactly one symbol definition**

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

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

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

main.c

main.o

Symbol table:
- sum: undefined
- aray: defined
- main: defined

Resolution result:
- sum: defined in sum.o

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

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

sum.c

sum.o

Symbol table:
- sum: defined

We learned about symbols and symbol table

So, how linker resolves duplicate symbol definitions?

(Background) Strong vs. Weak Symbols

- Program symbols are either strong or weak
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals
    - Or ones declared with specifier `extern`

```
int foo = 5; // Strong Symbol

int bar;     // Weak Symbol

void p1 () { // Strong Symbol
    /* ... */
}
```

How Linker Resolves Duplicate Symbol Definitions

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

```
int x;
void p1() {} // strong
```

a.c

```
void p1() {} //strong
```

b.c

Link time error:
two strong symbols (p1)

```
$ gcc a.c b.c

/usr/bin/ld: /tmp/ccPq104j.o: in function `p1':
b.c:(.text+0x0): multiple definition of `p1';
/tmp/cclnMRiY.o:a.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

How Linker Resolves Duplicate Symbol Definitions

- **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

```
int x=7; // strong
void p1() {}
```
a.c

```
int x; // weak
void p2() {}
```
b.c

References to x will refer to the same initialized variable

★ ***Linker does not do type checking*** →  ***Any problems?***

Memory Overwrite Bugs

```c
#include<stdio.h>
void foo(void);

int x = 12345; // Strong Symbol
int y = 12344;

int main() {
  foo();
  printf("%d\n", y);
  return 0;
}
```

main.c

```c
double x; // Weak Symbol

void foo(void) {
  x = 3.14;
}
```

variable.c

What gets printed?

```
$ gcc main.c variable.c -fcommon -o main
$ ./main
1074339512
```

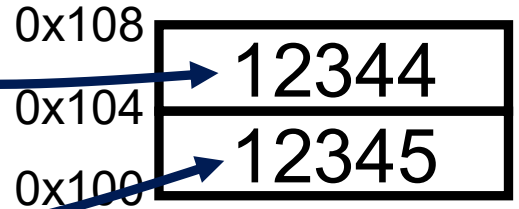
Memory Overwrite Bugs

```c
#include<stdio.h>
void foo(void);

int x = 12345; // Strong Symbol
int y = 12344;

int main() {
  foo();
  printf("%d\n", y);
  return 0;
}
```

main.c

```c
double x; // Weak Symbol

void foo(void) {
  x = 3.14;
}
```

variable.c



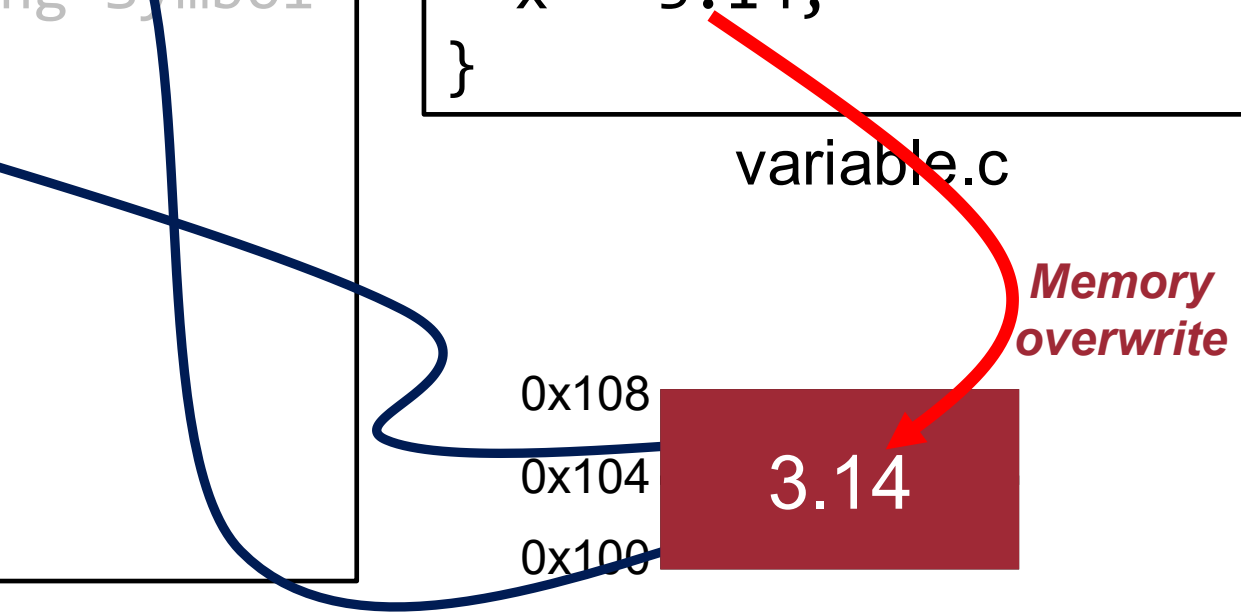
Memory Overwrite Bugs

```c
#include<stdio.h>
void foo(void);

int x = 12345; // Strong Symbol
int y = 12344;

int main() {
  foo();
  printf("%d\n", y);
  return 0;
}
```

main.c

```c
double x; // Weak Symbol

void foo(void) {
  x = 3.14;
}
```

variable.c



How Linker Resolves Duplicate Symbol Definitions

- **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

```
int x=7; // strong
void p1() {}
```
a.c

```
int x; // weak
void p2() {}
```
b.c

References to x will refer to the same initialized variable

★ ***Linker does not do type checking*** → ***Cause memory overwrite bugs***

How Linker Resolves Duplicate Symbol Definitions

- **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() {}  // strong
```

```
p1() {}  // strong
```

Link time error: two strong symbols (p1)

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

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

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

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

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

Writes to x in p2 might overwrite y by any chance! Evil!

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

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

Writes to x in p2 might overwrite y! Nasty!

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

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

References to x will refer to the same initialized variable.

Global Variables: Take Aways

- 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 not defined in some file

Linking Process

Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

main.c

main.o

Symbol table:
- sum: undefined
- aray: defined
- main: defined

Resolution result:
- sum: defined
  in sum.o

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

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

sum.c

sum.o

Symbol table:
- sum: defined

Step 2: Relocation

Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

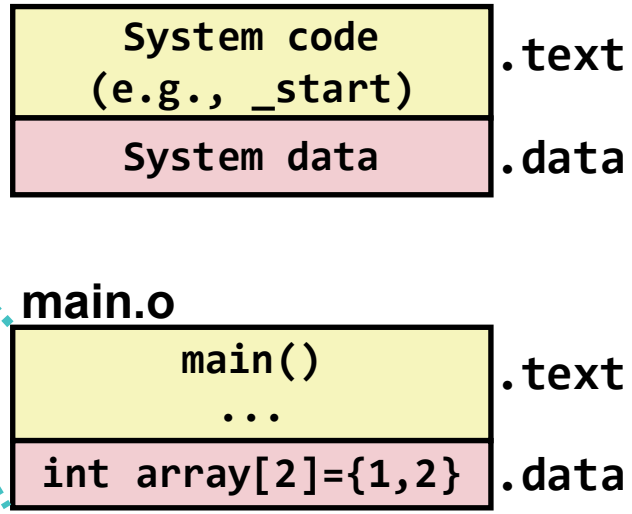
main.c

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

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

sum.c

main.c

sum.o



Linking-relevant information in relocatable object files

Step 2: Relocation

Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

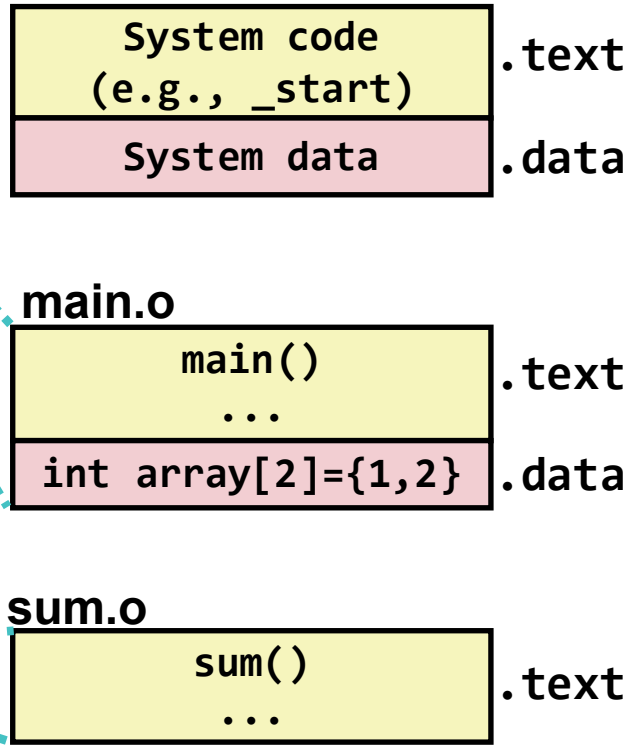
main.c

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

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

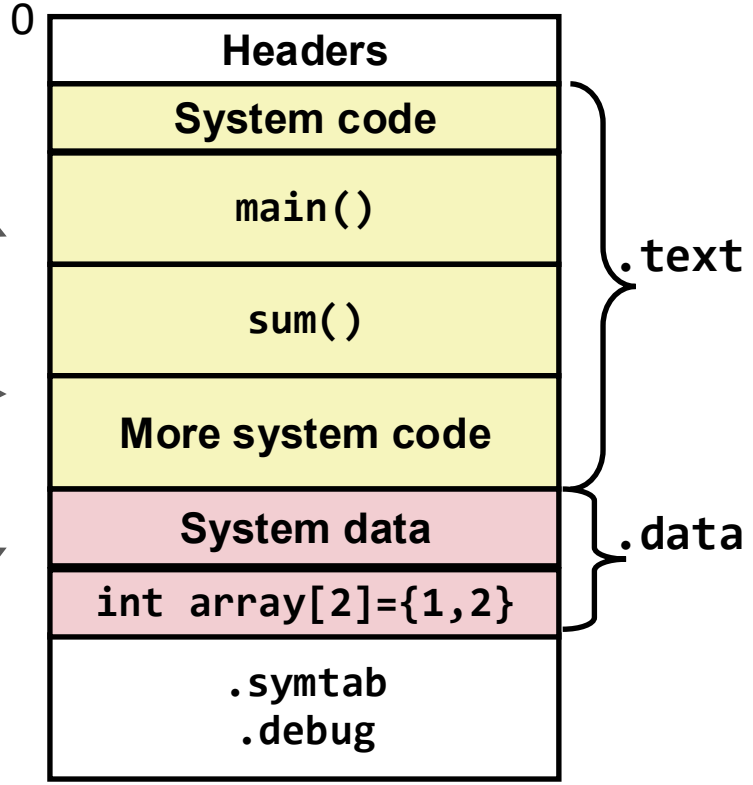
sum.c

main.c

sum.o



Linking-relevant information in relocatable object files



Executable Object File

** We assume static linking*

Step 2: Relocation

- The process of connecting symbolic references with symbolic definitions by adjusting address-dependent code to run at a specific memory location

Relocation Problems

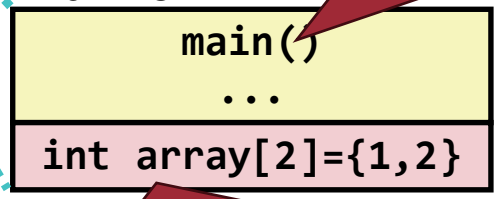
Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

main.c

main.c

main.o



At this stage, the linker does **not yet know the address of `sum()`** in the final executable.

Likewise, it does **not yet know where `array[]` will be placed in the `.data` section** of the final executable.

Relocation Problems

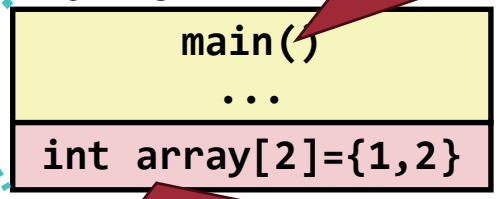
Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

main.c

main.c

main.o



At this stage, the linker does **not yet know the address of `sum()`** in the final executable.

Likewise, it does **not yet know where `array[]` will be placed in the `.data` section** of the final executable.

The exact locations of functions and data cannot be determined until linking is complete → ***How can the linker determine their locations?***

Relocation Entry

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

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

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

main.c

```
00 <main>:
  ...
  11:   bf 00 00 00 00          mov    edi,0x0
  16:   b8 00 00 00 00          mov    eax,0x0
  1b:   e8 00 00 00 00          call   20 <main+0x20>
  ...
```

main.o

Relocation Entry

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

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

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

main.c

```
00 <main>:
  ...
  11:   bf 00 00 00 00          mov     edi,0x0
  16:   b8 00 00 00 00          mov     eax,0x0
  1b:   e8 00 00 00 00          call    20 <main+0x20>
  ...
```

main.o

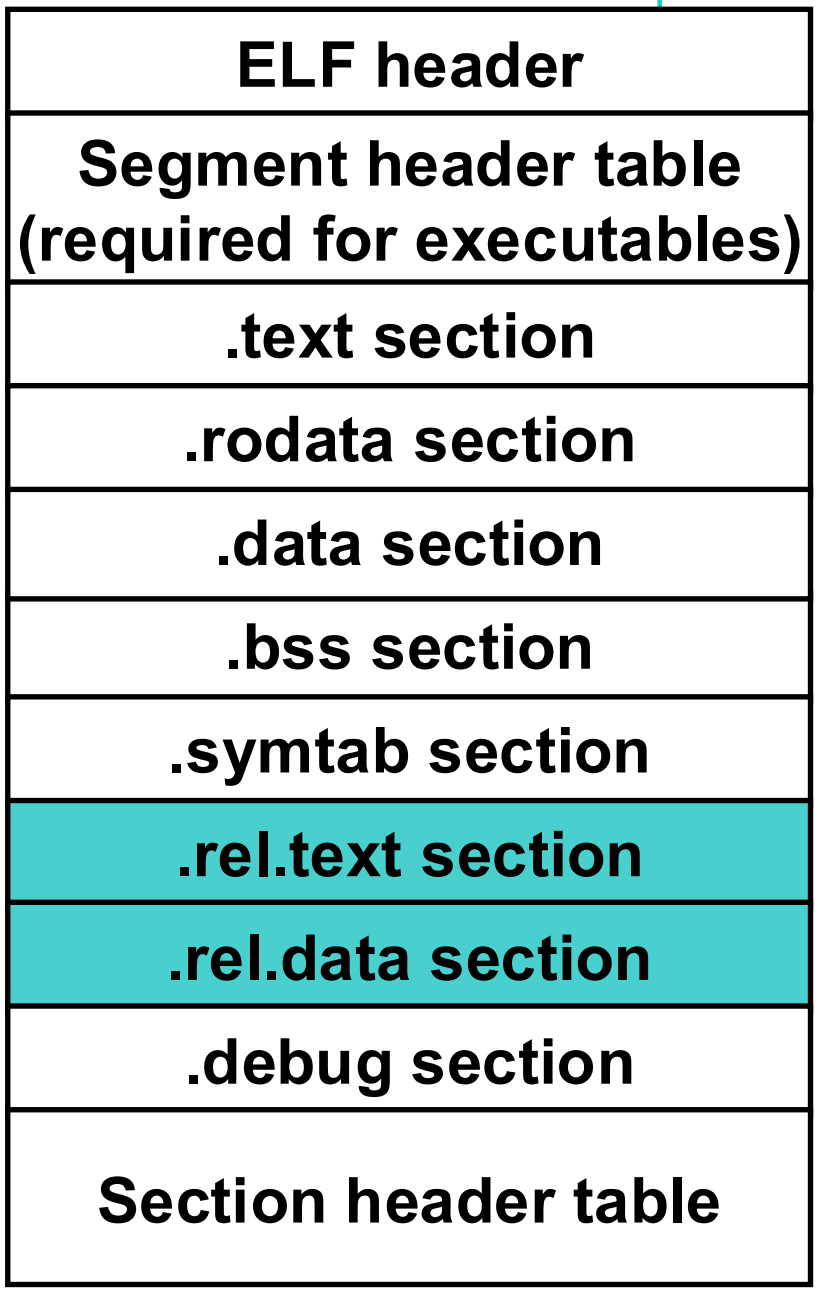
First argument: address of the **array** variable

Function address (**sum**'s address)

In relocatable object files, unresolved address fields are initially filled with 0. These spots are recorded as ***relocation entries***.

Relocation Entry (.rel.text, .rel.data)

- **.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)



Relocation Entry

- Tells the linker how a particular location should be modified when producing the executable object file

- Stored in **`.rel.text`** and **`.rel.data`** section

```
$ readelf -r <binary>
```

Relocation Entry

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

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

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

main.c

```
00 <main>:
  ...
  11:   bf 00 00 00 00          mov    edi,0x0
  16:   b8 00 00 00 00          mov    eax,0x0
  1b:   e8 00 00 00 00          call   20 <main+0x20>
  ...
```

main.o

First argument: address of the **array** variable

Function address (**sum**'s address)

```
$ readelf -r main.o

Relocation section '.rela.text' at offset 0x198 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000012  00030000000a R_X86_64_32       0000000000000000 array + 0
00000000001c  00050000004 R_X86_64_PLT32    0000000000000000 sum - 4
```

Relocation Entry

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

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

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

main.c

```
00 <main>:
  ...
  11:   bf 00 00 00 00          mov    edi,0x0
  16:   b8 00 00 00 00          mov    eax,0x0
  1b:   e8 00 00 00 00          call   20 <main+0x20>
  ...
```

main.o

First argument: address of the **array** variable

Function address (**sum**'s address)

Direct absolute address relocation

PC-relative relocation

```
$ readelf -r main.o

Relocation section '.rela.text' at off    0x198 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000012  00030000000a R_X86_64_32       0000000000000000 array + 0
00000000001c  000500000004 R_X86_64_PLT32    0000000000000000 sum - 4
```

Relocated .text Section


```
$ gcc -fno-pie -no-pie -o final main.o sum.o
```

```
401106 <main>:
  ...
  401117:    bf 10 40 40 00          mov    edi,0x404010
  40111c:    b8 00 00 00 00          mov    eax,0x0
  401121:    e8 08 00 00 00          call   40112e <sum>
  401126:    89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  401129:    8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  40112c:    c9                      leave
  40112d:    c3                      ret

000000000040112e <sum>:
  40112e:    f3 0f 1e fa             endbr64
  401132:    55                      push   rbp
  401133:    48 89 e5                mov    rbp,rsp
  401136:    48 89 7d e8             mov    QWORD PTR [rbp-0x18],rdi
  40113a:    89 75 e4                mov    DWORD PTR [rbp-0x1c],esi
```

Relocated .text Section

Absolute address: the linker determines the exact address of `array` in `.data` and simply writes that address into this location.

```
$ gcc -fno-pie -
```

```
401106 <main>:
  ...
  401117:    bf 10 40 40 00          mov    edi,0x404010
  40111c:    b8 00 00 00 00          mov    eax,0x0
  401121:    e8 08 00 00 00          call   40112e <sum>
  401126:    89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  401129:    8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  40112c:    c9                      leave
  40112d:    c3                      ret

000000000040112e <sum>:
  40112e:    f3 0f 1e fa             endbr64
  401132:    55                      push   rbp
  401133:    48 89 e5                mov    rbp,rsp
  401136:    48 89 7d e8             mov    QWORD PTR [rbp-0x18],rdi
  40113a:    89 75 e4                mov    DWORD PTR [rbp-0x1c],esi
```

Relocated .text Section

Absolute address: the linker determines the exact address of `array` in `.data` and simply writes that address into this location.

```
$ gcc -fno-pie -
```

```
401106 <main>:
  ...
  401117:     bf 10 40 40 00          mov    edi,0x404010
  40111c:     b8 00 00 00 00          mov    eax,0x0
  401121:     e8 08 00 00 00          call   40112e <sum>
  401126:     89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  401129:     8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  40112c:     c9
  40112d:     c3
  0x8
000000000040112e <sum>:
  40112e:     f3 0f 1e fa
  401132:     55                      push   rbp
  401133:     48 89 e5                mov    rbp,rsp
  401136:     48 89 7d e8             mov    QWORD PTR [rbp-0x18],rdi
  40113a:     89 75 e4                mov    DWORD PTR [rbp-0x1c],esi
```

PC-relative address:

Target Address = RIP + [address part]

0x40112e = 0x401126 + 0x8

Linking Process

Step 1: Symbol Resolution → **Step 2**: Relocation

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

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

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

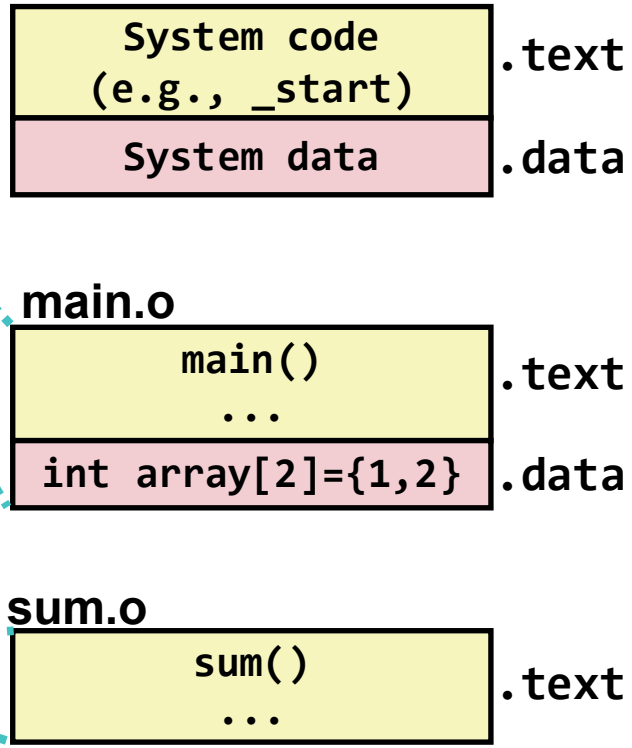
main.c

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

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

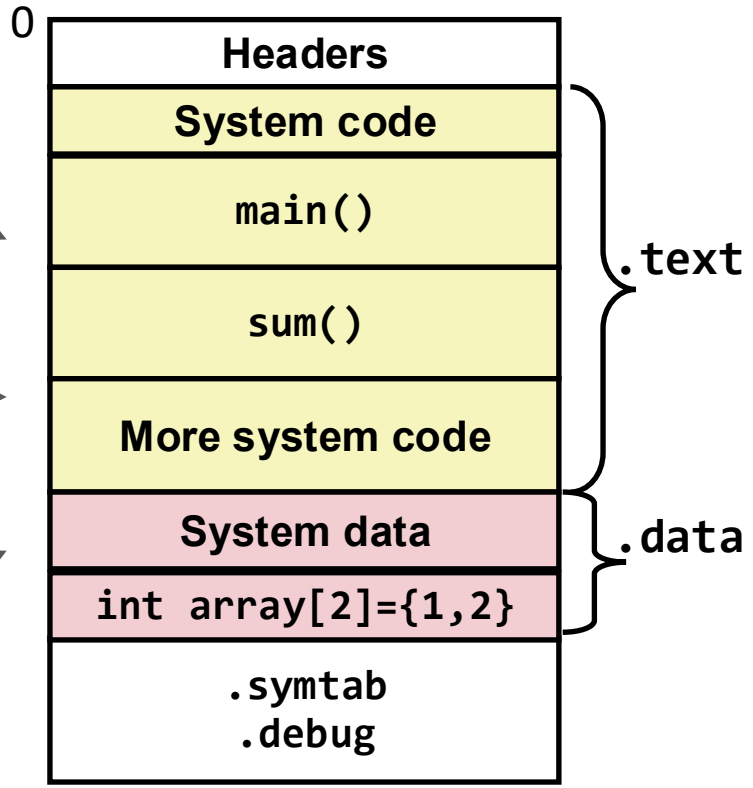
sum.c

main.c

sum.o



Linking-relevant information in relocatable object files



Executable Object File

** We assume static linking*

Question?