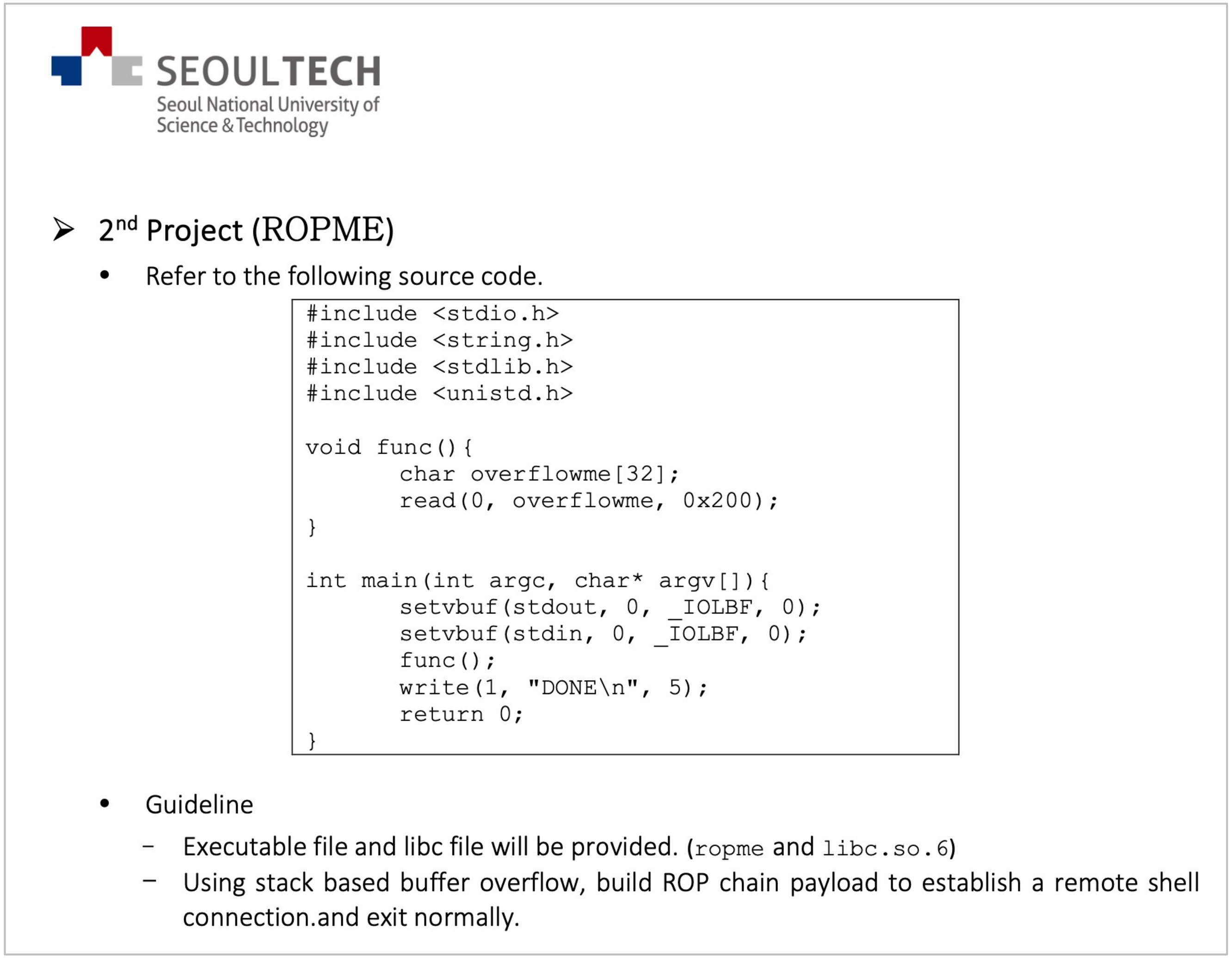
Linking

Computer Systems
Friday, November 22, 2024

Team Project

■ ROPME v1.2



SEOULTECH
Seoul National University of Science & Technology

➢ 2nd Project (ROPME)

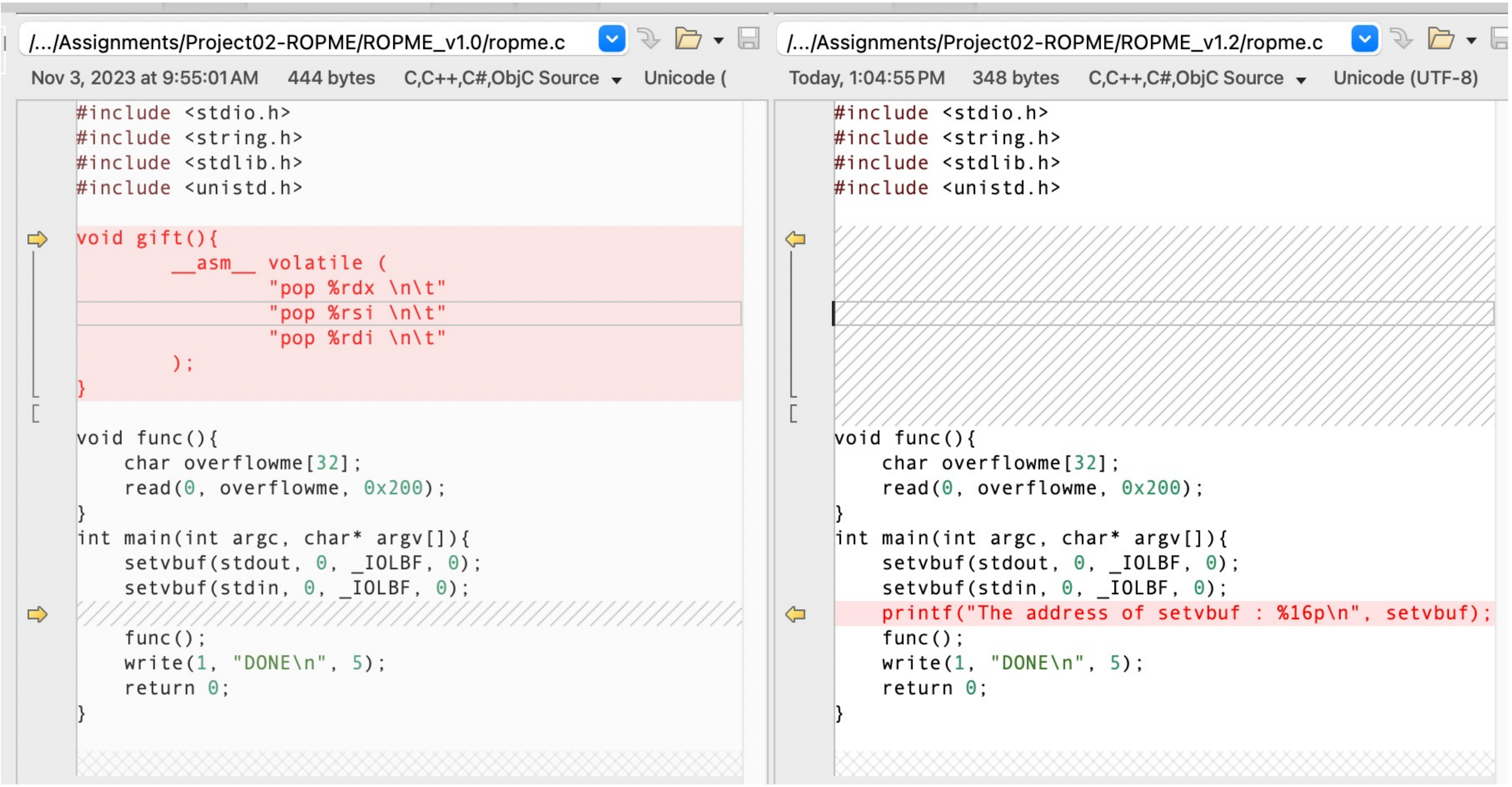
- Refer to the following source code.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

void func(){
       char overflowme[32];
       read(0, overflowme, 0x200);
}

int main(int argc, char* argv[]){
       setvbuf(stdout, 0, _IOLBF, 0);
       setvbuf(stdin, 0, _IOLBF, 0);
       func();
       write(1, "DONE\n", 5);
       return 0;
}
```

- Guideline
  - Executable file and libc file will be provided. (`ropme` and `libc.so.6`)
  - Using stack based buffer overflow, build ROP chain payload to establish a remote shell connection.and exit normally.

Team Project

- **ROPME v1.2**



/.../Assignments/Project02-ROPME/ROPME_v1.0/ropme.c

Nov 3, 2023 at 9:55:01 AM   444 bytes   C,C++,C#,ObjC Source   Unicode (

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

void gift(){
        __asm__ volatile (
                "pop %rdx \n\t"
                "pop %rsi \n\t"
                "pop %rdi \n\t"
        );
}

void func(){
    char overflowme[32];
    read(0, overflowme, 0x200);
}
int main(int argc, char* argv[]){
    setvbuf(stdout, 0, _IOLBF, 0);
    setvbuf(stdin, 0, _IOLBF, 0);
    func();
    write(1, "DONE\n", 5);
    return 0;
}
```

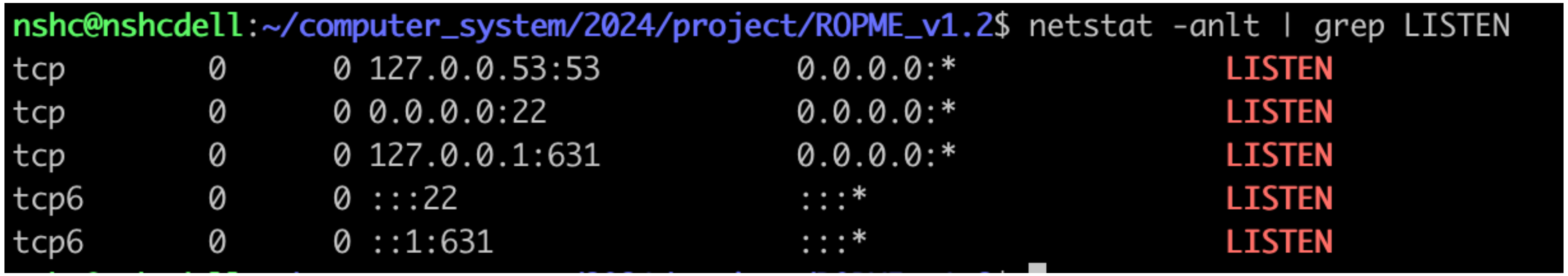
/.../Assignments/Project02-ROPME/ROPME_v1.2/ropme.c

Today, 1:04:55 PM   348 bytes   C,C++,C#,ObjC Source   Unicode (UTF-8)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

void func(){
    char overflowme[32];
    read(0, overflowme, 0x200);
}
int main(int argc, char* argv[]){
    setvbuf(stdout, 0, _IOLBF, 0);
    setvbuf(stdin, 0, _IOLBF, 0);
    printf("The address of setvbuf : %16p\n", setvbuf);
    func();
    write(1, "DONE\n", 5);
    return 0;
}
```

Team Project

- **ROPME v1.2**
  - How to debug
    - break at `ret`
  - Socket (run.sh)
    - TCP Port open for : 31000 ~ 32000
    - Avoid binding error : 31xxx



```
nshc@nshcdell:~/computer_system/2024/project/ROPME_v1.2$ netstat -anlt | grep LISTEN
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
tcp6       0      0 ::1:631                 :::*                    LISTEN
```

Today

- **Linking**
  - Motivation
  - What it does
  - How it works

Example C Program

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

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

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

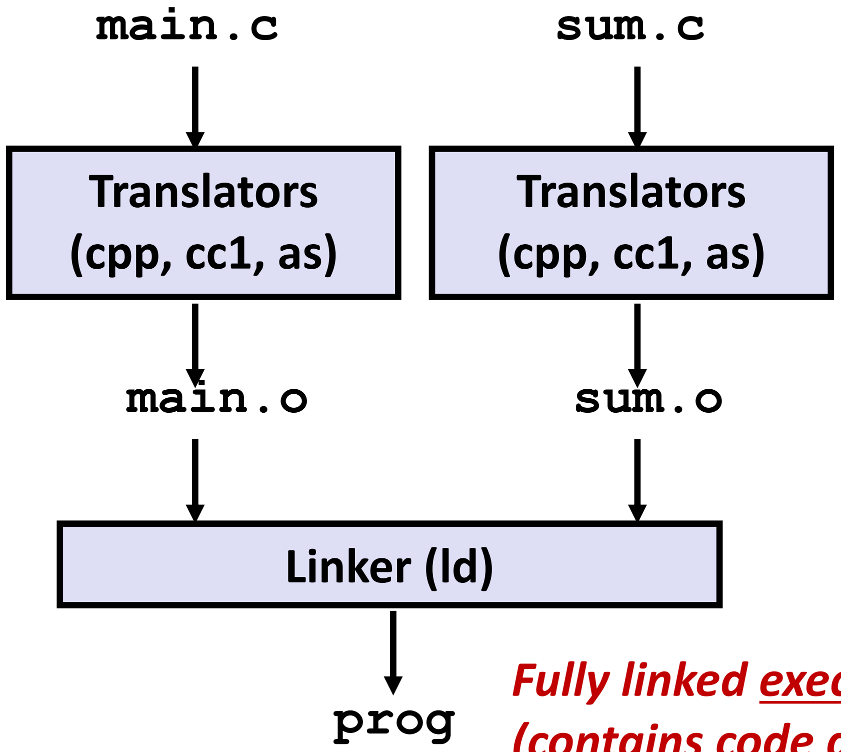
main.c

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

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

sum.c

Linking

- **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`)

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
    - Header files in C declare types that are defined in libraries

Why Linkers? (cont)

- **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...
    - **Option 1: *Static Linking***
      - Executable files and running memory images contain only the library code 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

What Do Linkers Do?

- **Step 1: Symbol resolution**

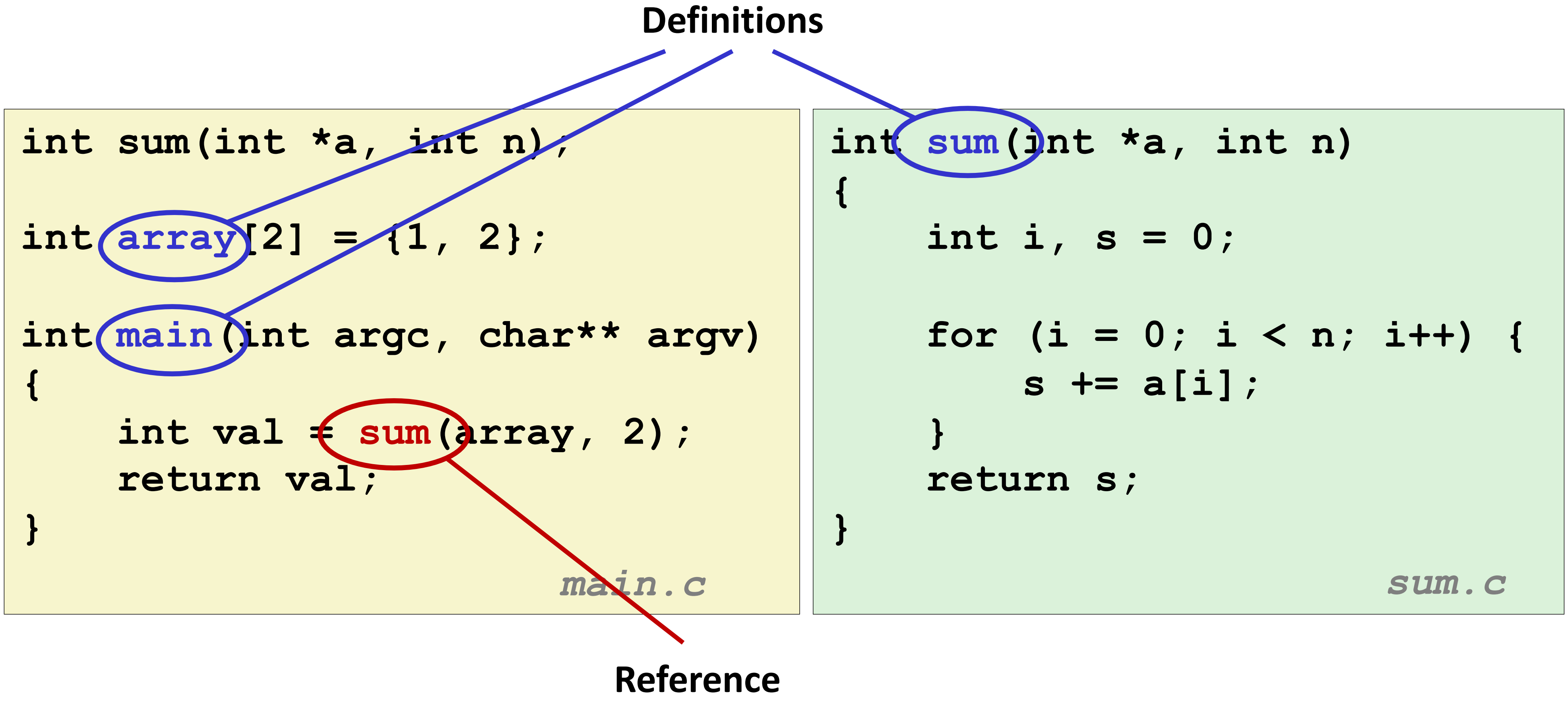
  - Programs define and reference *symbols* (global variables and functions):

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

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

  - **During symbol resolution step, the 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? (cont'd)

- **Step 2: Relocation**
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Let's look at these two steps in more detail….

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)

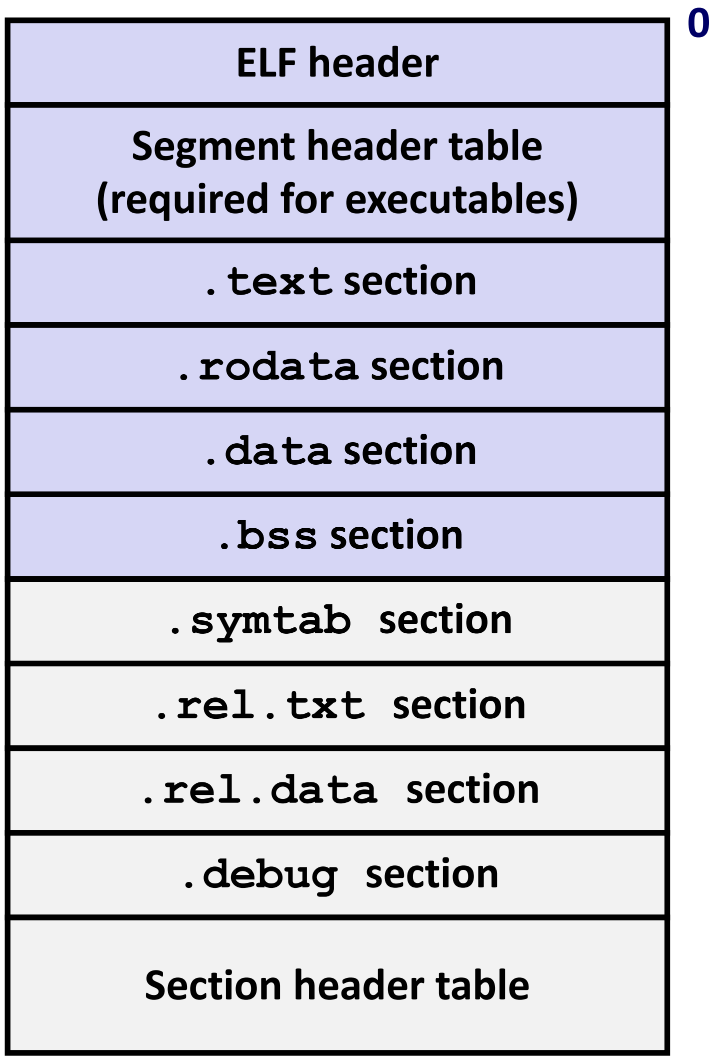
- **Standard binary format for object files**

- **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.
- **Segment header table**
  - Page size, virtual address memory segments (sections), segment sizes.
- **`.text` section**
  - Code
- **`.rodata` section**
  - Read only data: jump tables, string constants, ...
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space



0

ELF header

Segment header table (required for executables)

.text section

.rodata section

.data section

.bss section

.symtab section

.rel.txt section

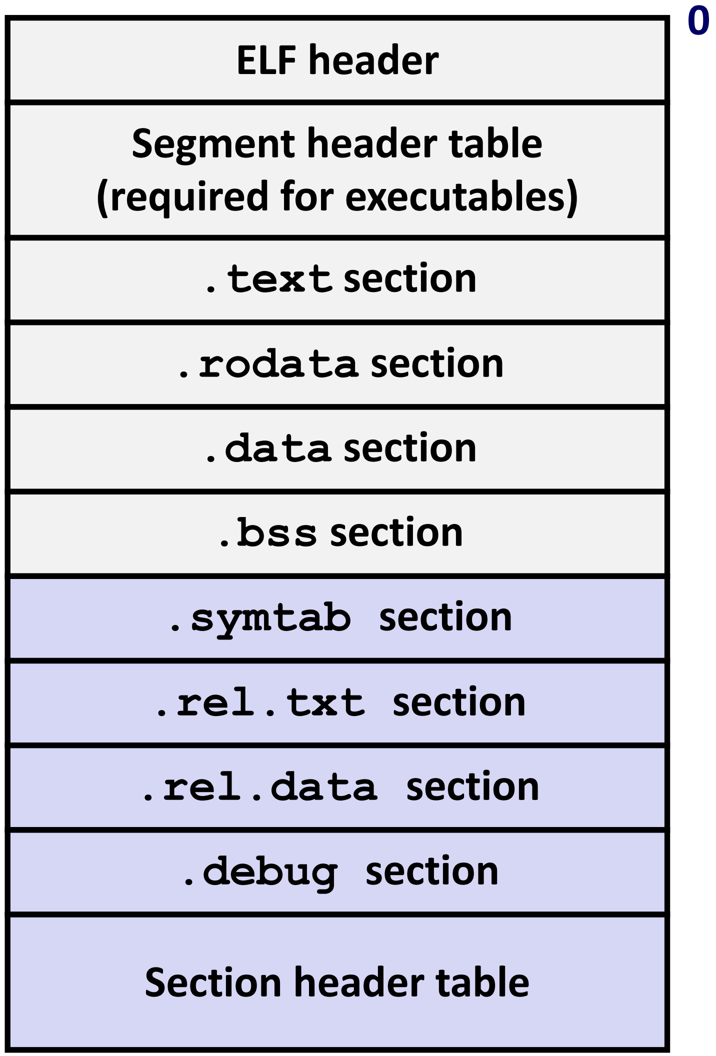
.rel.data section

.debug section

Section header table

ELF Object File Format (cont.)

- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying
- `.rel.data` section
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging (`gcc -g`)
- Section header table
  - Offsets and sizes of each section



0

ELF header

Segment header table (required for executables)

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.rel.txt` section

`.rel.data` section

`.debug` section

Section header table

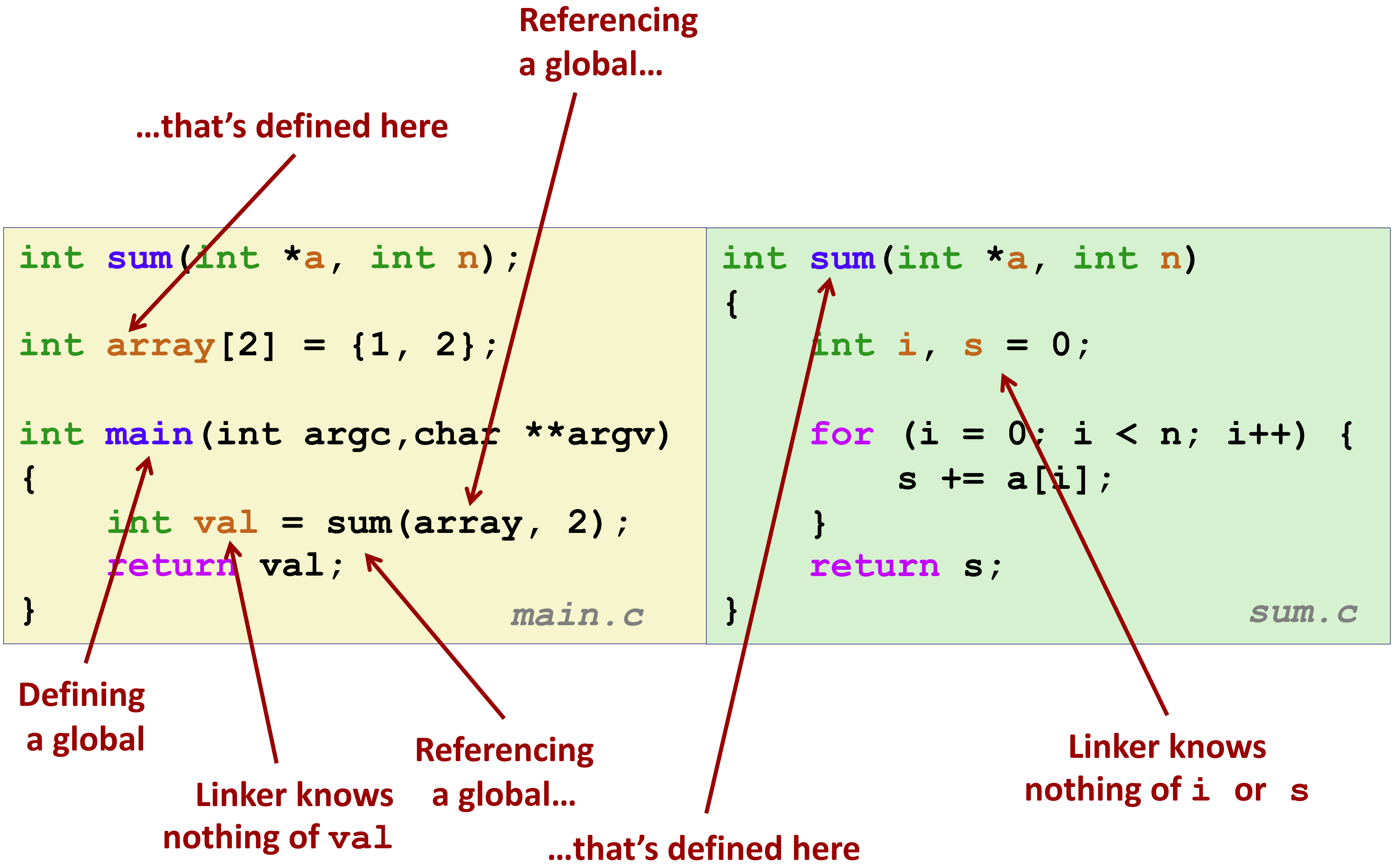
Linker Symbols

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

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

- **Local symbols**
  - Symbols that are defined and referenced exclusively by module *m*.
  - e.g, C functions and global variables defined with the `static` attribute.
  - **Local linker symbols are *not* local program variables**

Step 1: Symbol Resolution



```
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

Referencing a global…

…that's defined here

Defining a global

Linker knows nothing of `val`

Referencing a global…

…that's defined here

Linker knows nothing of `i` or `s`

Symbol Identification

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

symbols.c:

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

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

Names:

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

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

Local Symbols

- **Local non-static C variables vs. local static C variables**
  - Local non-static C variables: stored on the stack
  - Local static C variables: stored in either `.bss` or `.data`

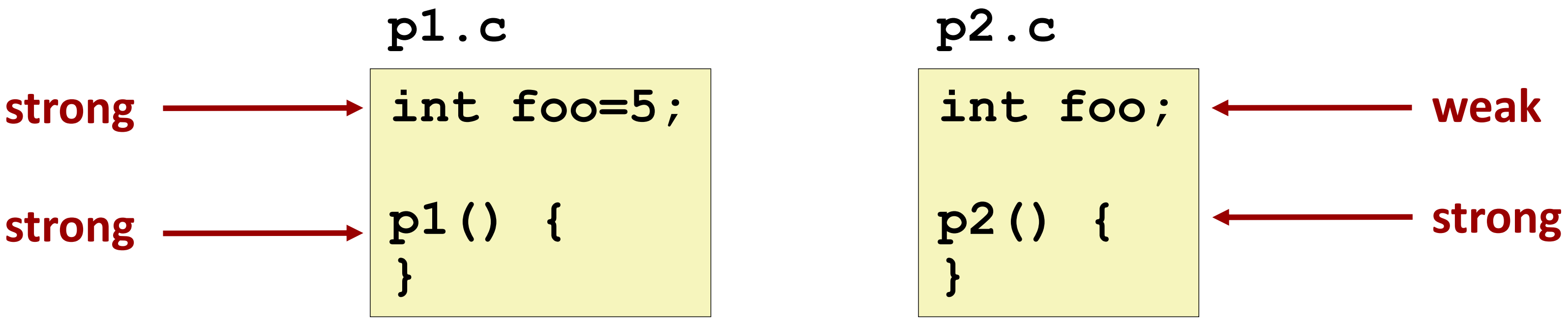
```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}          static-local.c
```

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

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

How Linker Resolves Duplicate Symbol Definitions

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



`p1.c`

```
int foo=5;

p1() {
}
```

`p2.c`

```
int foo;

p2() {
}
```

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`
- **Puzzles on the next slide**

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 might overwrite y! Nasty!

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

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

References to x will refer to the same initialized variable.

Important: Linker does not do type checking.

Type Mismatch Example

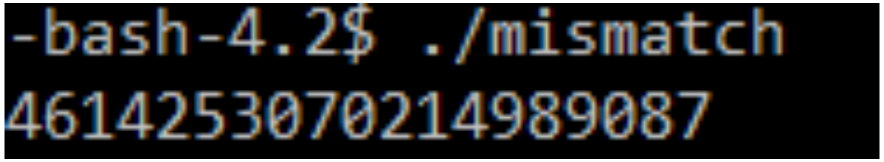
```
long int x;  /* Weak symbol */

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?



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

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 not defined in some file

Linking Example

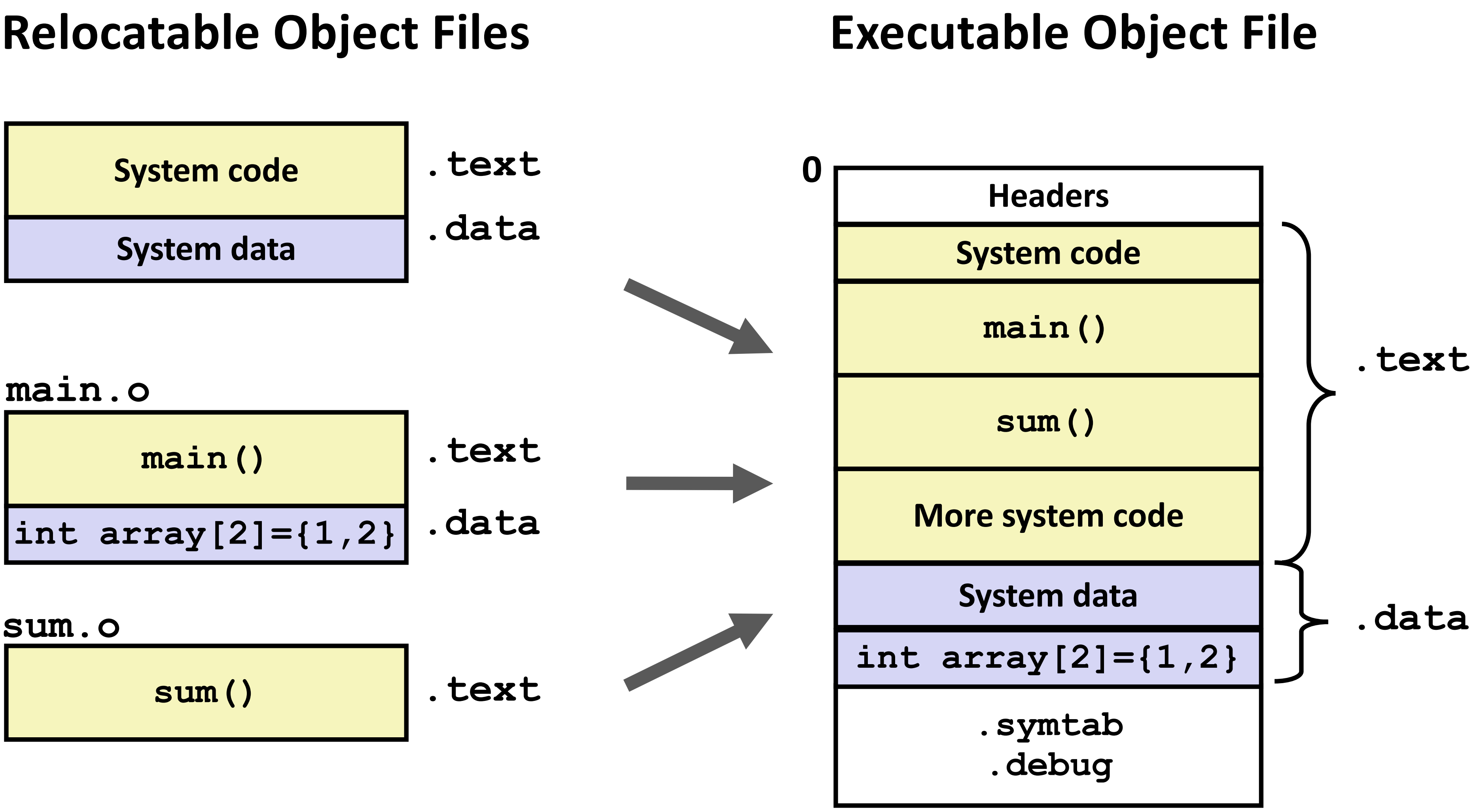
```c
int sum(int *a, int n);

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

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

main.c

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

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

sum.c

Step 2: Relocation



Relocation Entries

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

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

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

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

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

Relocated .text section

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

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

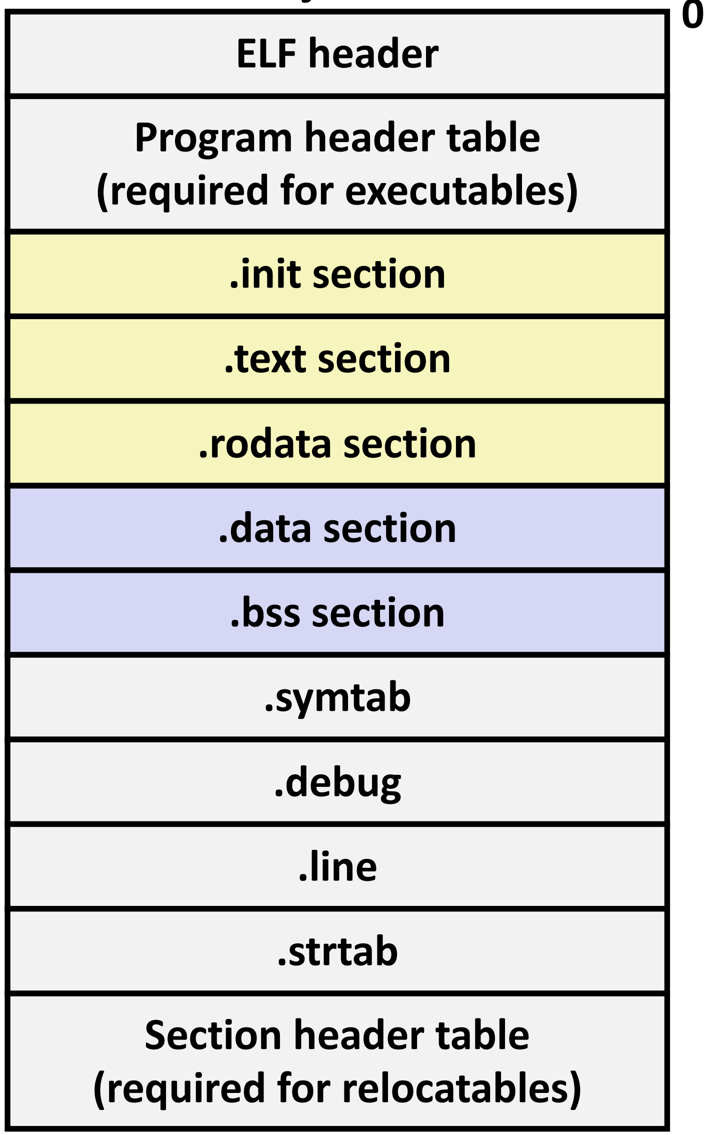
`callq` instruction uses PC-relative addressing for sum():

`0x4004e8` = `0x4004e3` + `0x5`

Source: `objdump -d prog`

Loading Executable Object Files

Executable Object File



0

ELF header

Program header table
(required for executables)

.init section

.text section

.rodata section

.data section

.bss section

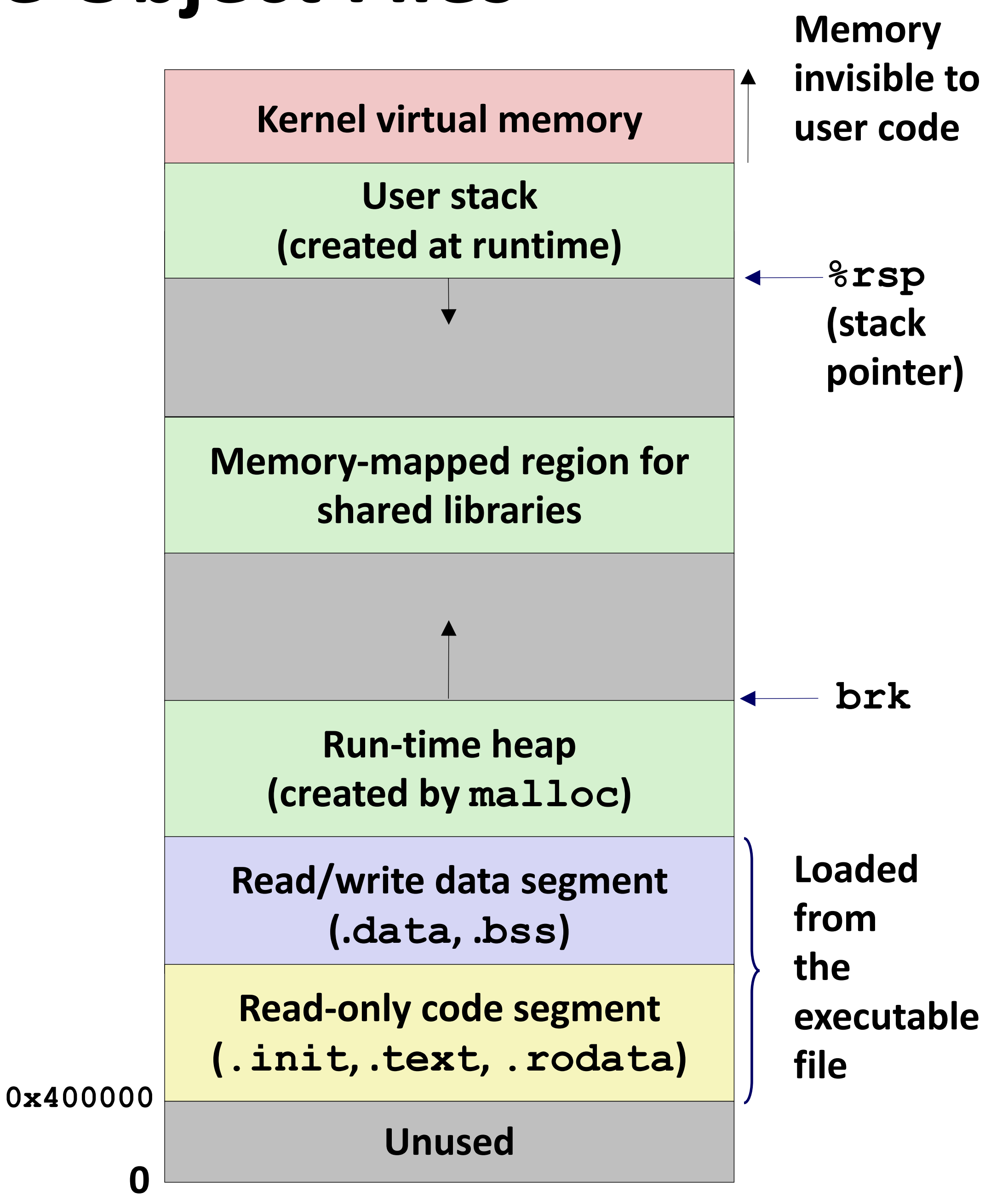
.symtab

.debug

.line

.strtab

Section header table
(required for relocatables)



Kernel virtual memory

Memory invisible to user code

User stack
(created at runtime)

`%rsp` (stack pointer)

Memory-mapped region for shared libraries

`brk`

Run-time heap
(created by `malloc`)

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

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

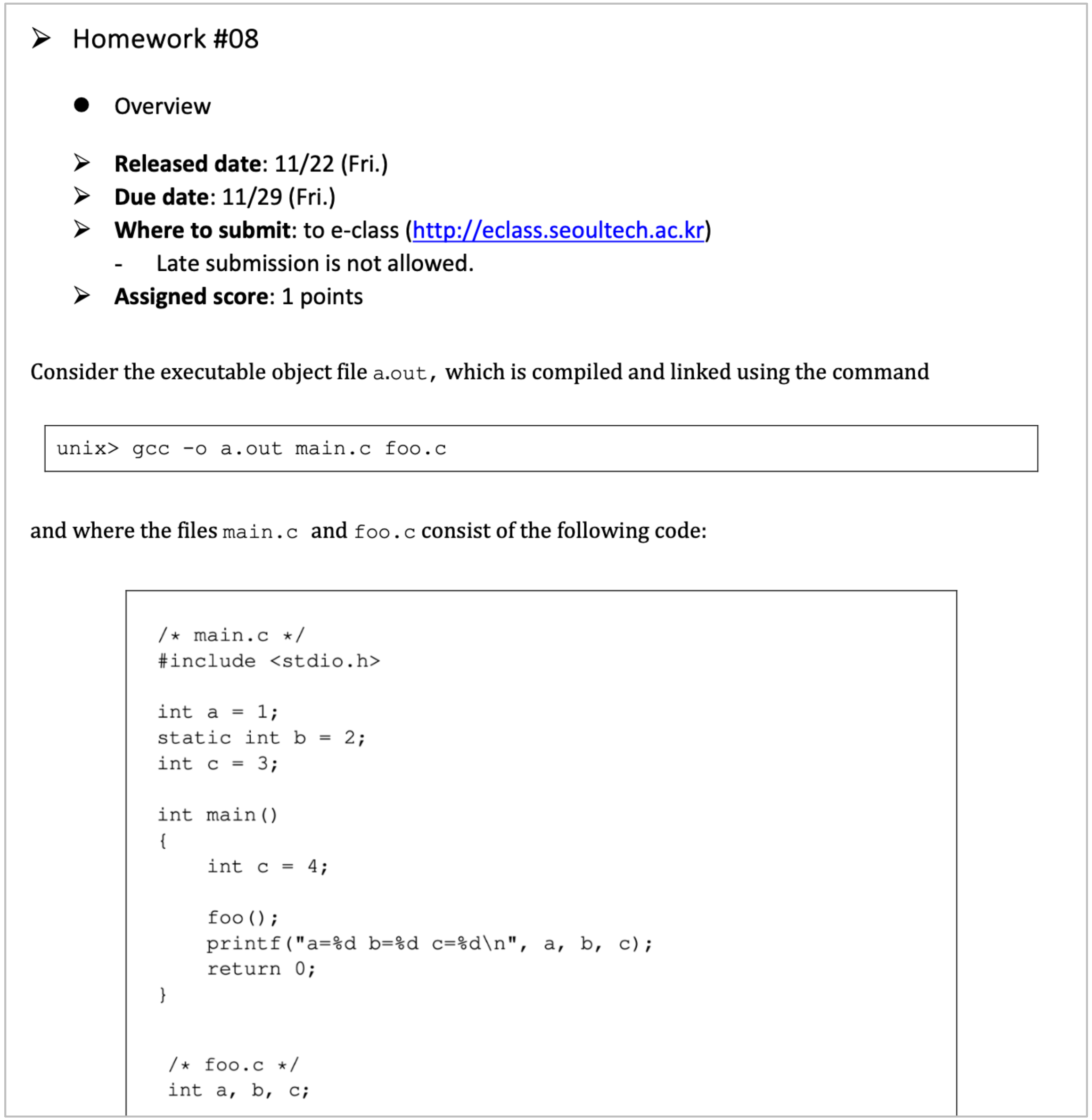
Loaded from the executable file

`0x400000`

Unused

0

Homework 8



- Homework #08

  - Overview

  - **Released date**: 11/22 (Fri.)
  - **Due date**: 11/29 (Fri.)
  - **Where to submit**: to e-class (http://eclass.seoultech.ac.kr)
    - Late submission is not allowed.
  - **Assigned score**: 1 points

Consider the executable object file a.out, which is compiled and linked using the command

```
unix> gcc -o a.out main.c foo.c
```

and where the files main.c and foo.c consist of the following code:

```
/* main.c */
#include <stdio.h>

int a = 1;
static int b = 2;
int c = 3;

int main()
{
    int c = 4;

    foo();
    printf("a=%d b=%d c=%d\n", a, b, c);
    return 0;
}


 /* foo.c */
 int a, b, c;
```

Linking Recap

- Usually: Just happens, no big deal
- Sometimes: Strange errors