UNM SCHOOL of ENGINEERING
Department of Computer Science

Lecture #21 or 22:
Chap. 7: Linking (and Loading)

Prof. Soraya Abad-Mota, PhD

Topics

- Chapter 7: Linking
  - Introduction: linkers and why study them, linking process
  - ELF format, linker symbols.
  - The linking process in more detail

Example C Program

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

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

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

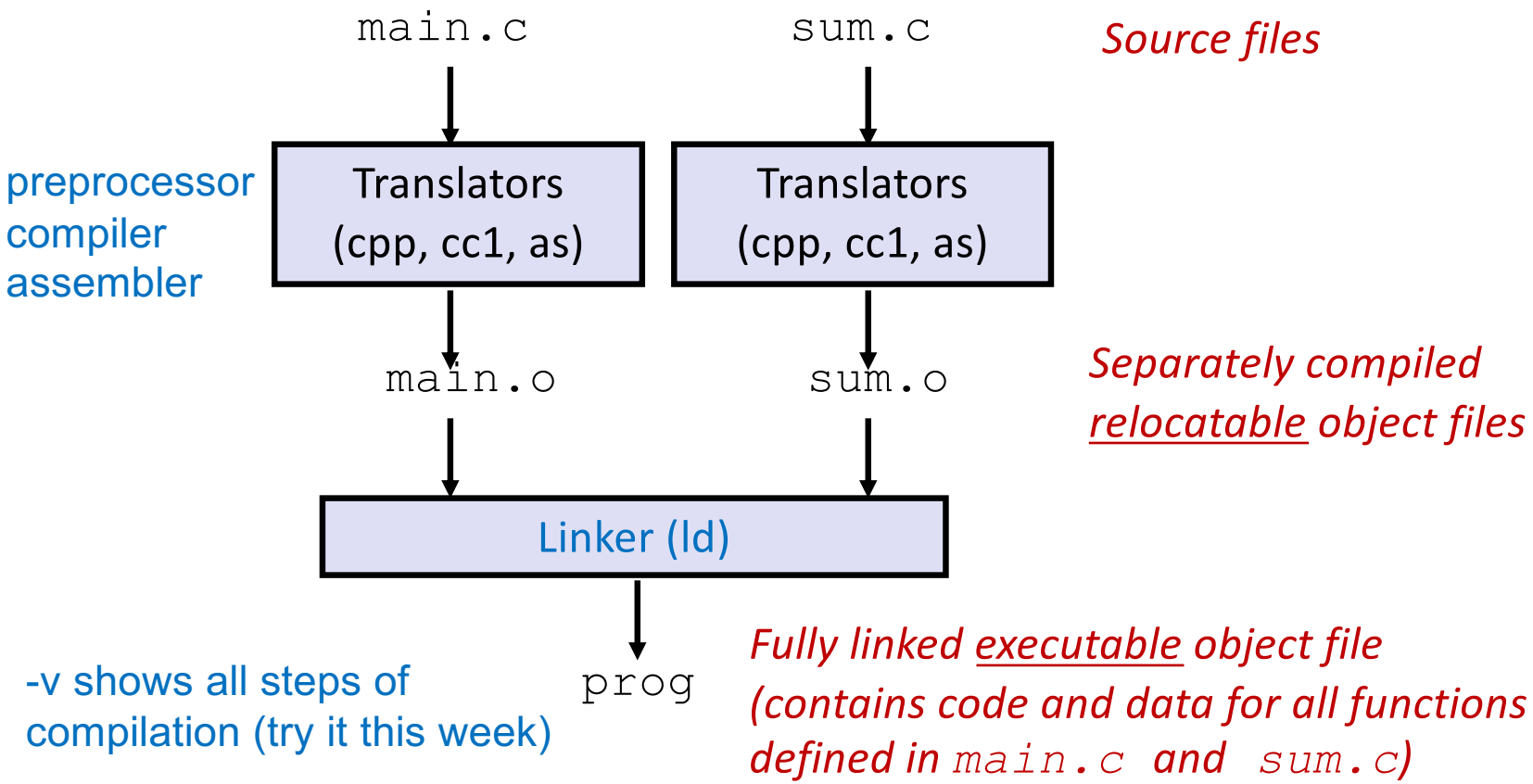
main.c

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

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

sum.c

Static Linking (gcc is a compiler driver)

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

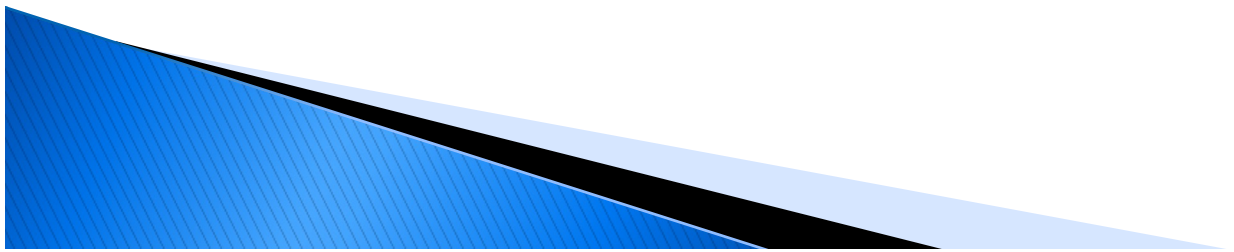


Linking

- *"Process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed."* p. 670
- This process can be perfomed: at compile time, at load time, and at run time.
- In *ancient* times it was performed manually.
- Today it is performed automatically by a program called *linker*.

Why Linkers? (crucial role)

- 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 (more on this later)
    - 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.
    - Yet executable files and running memory images contain only code for the functions they actually use.

Why study linkers?

- Understanding them helps: (p. 670)
  1. build large programs (linker errors: missing modules or libraries or incompatible library versions);
  2. avoid dangerous programming errors (e.g. multiply defined global variables);
  3. implementation of language scoping rules (e.g. global vs local variables, static functions);
  4. understand other systems concepts (e.g. virtual memory);
  5. enable you to exploit shared libraries (e.g. dynamic linking to serve dynamic content).

Input to a static linker

- Code and data sections
- Sections: (separate)
  - instructions
  - initialized global variables
  - uninitialized variables

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 the *symbol table*.
    - Symbol table is an array of `structs`
    - Each entry includes name, size, and location of symbol.
  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

What Do Linkers Do? (cont)

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

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

Three Kinds of Object Files (Modules)

(section 7.3)

- 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 load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

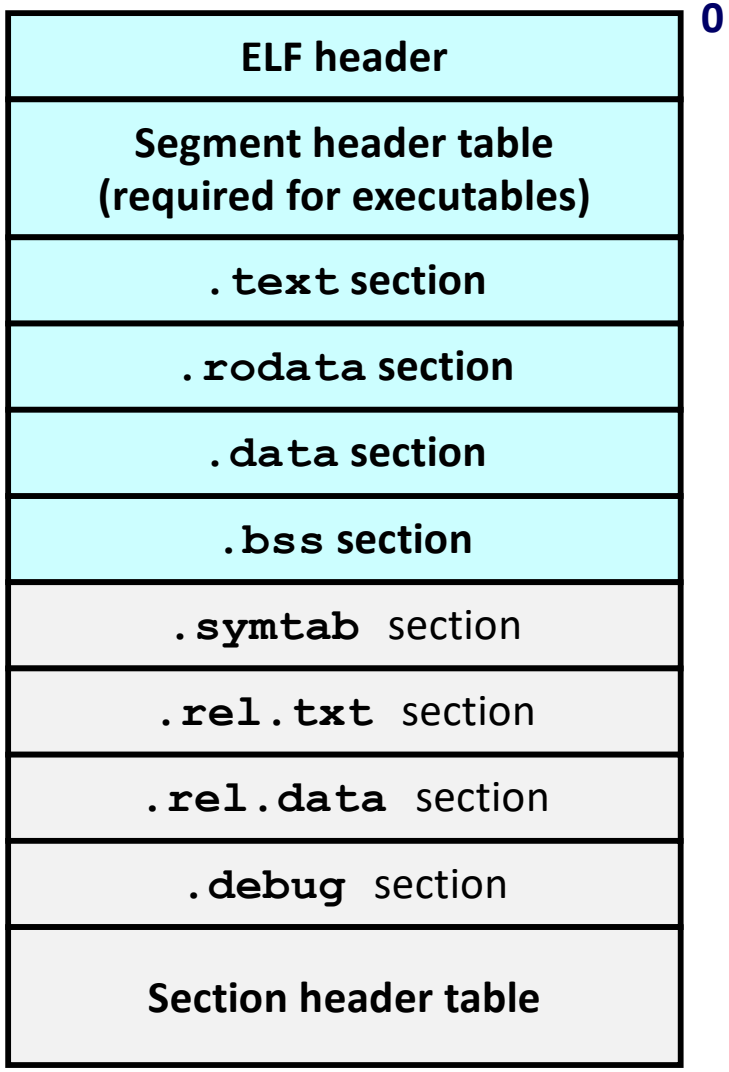
Executable and Linkable Format (ELF)

- Standard binary format for *object files* (= *object module*)
- One unified format for (sequences of bytes stored in disk files)
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)
- Generic name: ELF binaries

- Other OSs have other formats but concepts are the same (universal)

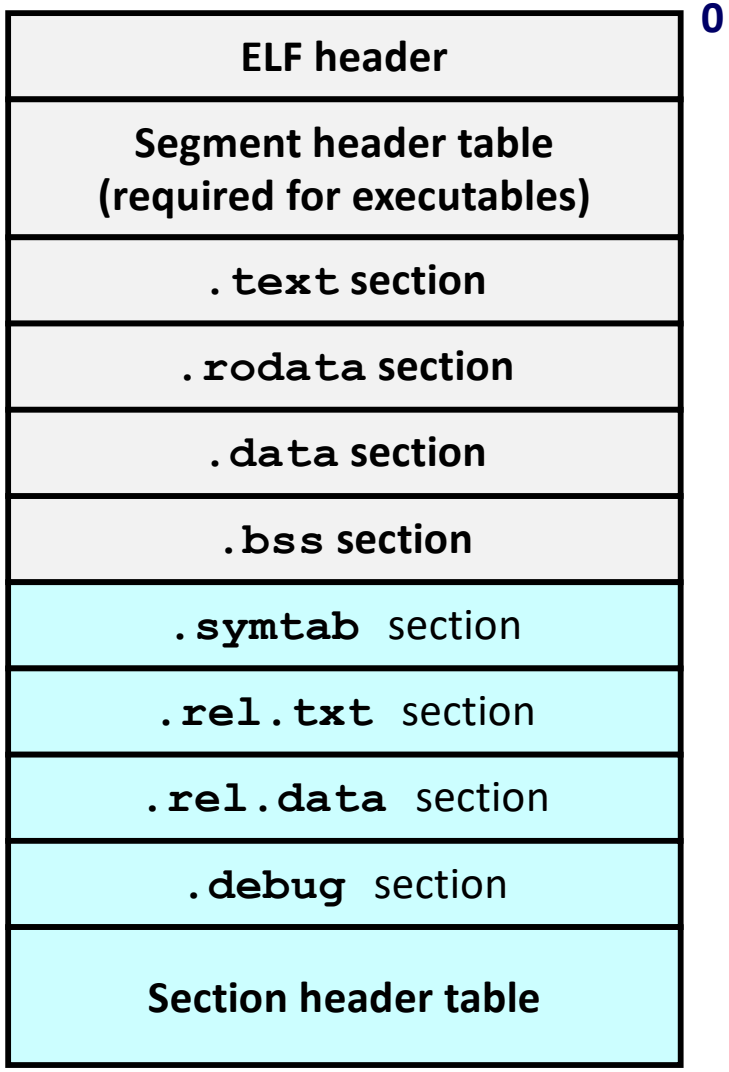
ELF Object File Format (section 7.4)

- Elf header
  - 16 bytes: Word size & byte ordering + file type (.o, exec, .so), size of header, machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes (for each section in obj.)
- `.text` section
  - machine code of compiled program
- `.rodata` section
  - Read only data: jump tables, format strings printf
- `.data` section
  - Initialized global and static C variables
- `.bss` section
  - Uninitialized global and static C variables
  - "Block Started by Symbol" "Better Save Space"
  - occupies no space (placeholder)



ELF Object File Format (cont.)

- `.symtab` section
  - Symbol table (no entries for local vars)
  - 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`**)
- `.line` (line #s map) and `.strtab` (string table)
- Section header table
  - Offsets and sizes of each section



Previously

- Intro to Linking: why study it
- Tasks of the linker:
  1. Symbol Resolution
  2. Relocation
- Types of object files
- ELF object file format

Linker Symbols (section 7.5)

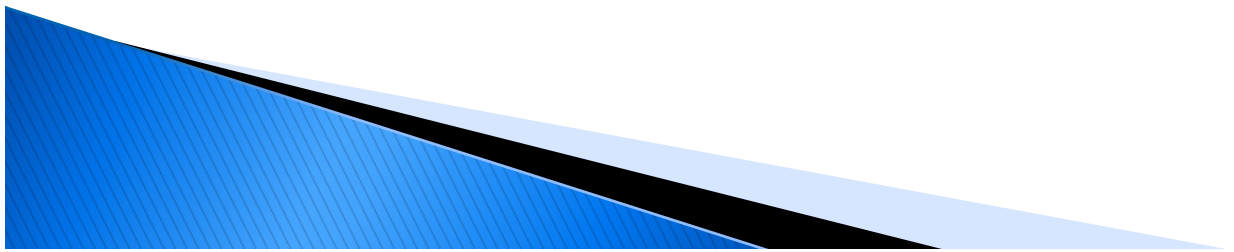
- Global symbols
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-**`static`** C functions and non-**`static`** global variables.
- External symbols
  - Global symbols that are referenced by module *m* but defined by some other module.
- Local symbols
  - Symbols that are defined and referenced exclusively by module *m*.
  - C functions and global variables defined with the **`static`** attribute.
  - **Local linker symbols are *not* local program variables** (local nonstatic program variables are managed on the stack, no interest to linker)

Topics (2)

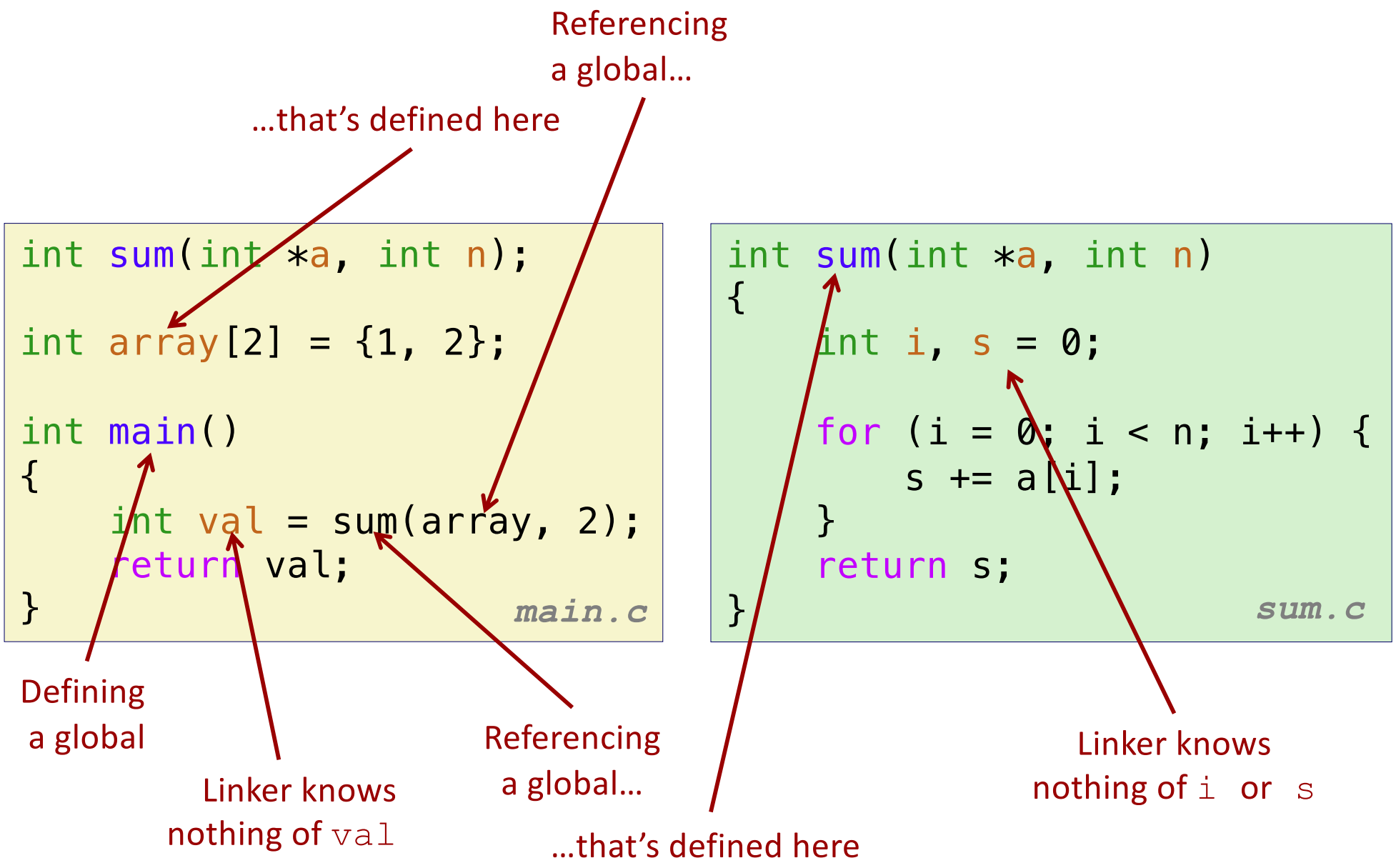
- Chapter 7: Linking
  - ELF format, linker symbols.
  - The linking process in more detail
    - Symbol resolution
    - Relocation

Linking process (1)

- Symbol resolution
  - associates each *symbol reference* with exactly one *symbol definition* from the symbol tables of its input relocatable object files.
  - Straightforward for: references to local symbols, defined in the same module as the reference.
  - Trickier for global symbols.



Step 1: Symbol Resolution (intro sec. 7.6)



An example

- Practice problem 7.1 p. 678
  - provides two .c files (`m.c` and `swap.c`)
  - variables and functions defined on each
  - for each symbol, the exercise asks for the type, if it has an entry in .symtab, where it is defined, and the section it belongs to.
  - uses definitions on section 7.5
  - Solution on p. 717, but try it out first

(a) m.c — *code/link/m.c*

```
1   void swap();
2
3   int buf[2] = {1, 2};
4
5   int main()
6   {
7       swap();
8       return 0;
9   }
```

code/link/m.c

(b) swap.c — *code/link/swap.c*

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   int *bufp1;
5
6   void swap()
7   {
8       int temp;
9
10      bufp1 = &buf[1];
11      temp = *bufp0;
12      *bufp0 = *bufp1;
13      *bufp1 = temp;
14  }
```

code/link/swap.c

Figure 7.5 **Example program for Practice Problem 7.1.**

Declaration and definition in C

- For functions:
  - Declaration: signature (prototype) only (by default: `extern`)
  - Definition: body of the function
- For variables:
  - Declaration: only the type, no memory allocation
    - Attained with `extern int var;`
  - Definition: type and memory allocation
    - E.g. `int var;`
    - Those declared as extern must be defined somewhere

Answers here

Symbol	.symtab entry?	Symbol type	Module where defined	Section
buf				
bufp0				
bufp1				
swap				
temp				

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`

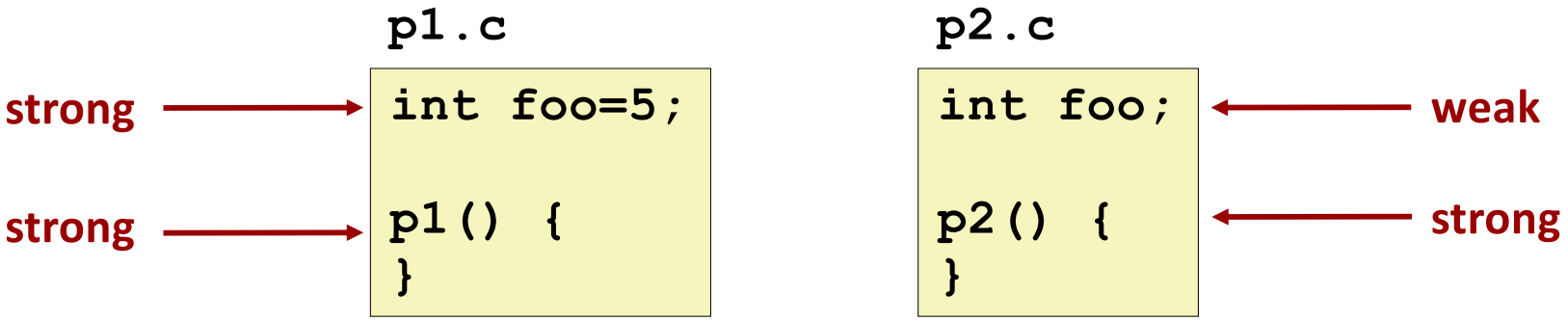
```
int f()
{
    static int x = 0;
    return x;
}

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

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

Creates local symbols in the symbol table with unique names, e.g., `x.1` (x in f) and `x.2` (x in g).

How Linker Resolves Duplicate Symbol Definitions

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



(Linux) Linker's Symbol Rules

- Rule 1: Multiple strong symbols[1] are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error
- Rule 2: Given a strong symbol and multiple weak symbols[1], choose the strong symbol
  - References to the weak symbol resolve to the strong symbol
- Rule 3: If there are multiple weak symbols[1], pick an arbitrary one
  - Can override this with **`gcc -fno-common`** (triggers error if encounters multiply defined global symbols) or –Werror (to turn warnings into errors)

[1] with the same name

Topics (3)

- Chapter 7: Linking
  - ELF format, linker symbols.
  - The linking process in more detail
    - Linker puzzles
    - The relocation phase of the Linking process

A note about

- Static global variables
- A reflection on what is stored in the object files vs what's stored in the stack.

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() {}
```