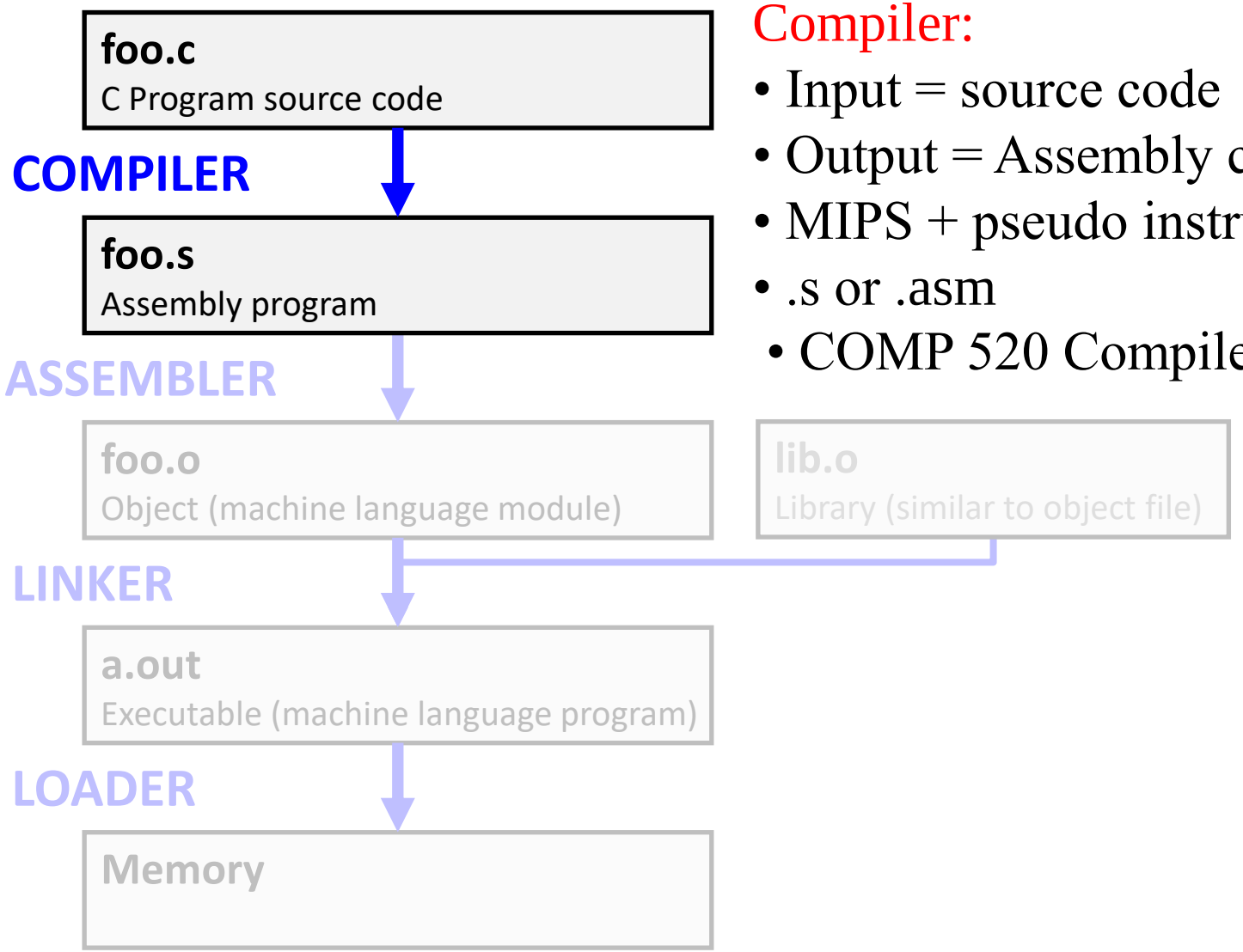
Starting a Program

Outline

- Compiler
- Assembler
- Linker
- Loader
- Example

Steps to Starting a Program



Compiler:

- Input = source code
- Output = Assembly code
- MIPS + pseudo instructions
- .s or .asm
- COMP 520 Compiler Design

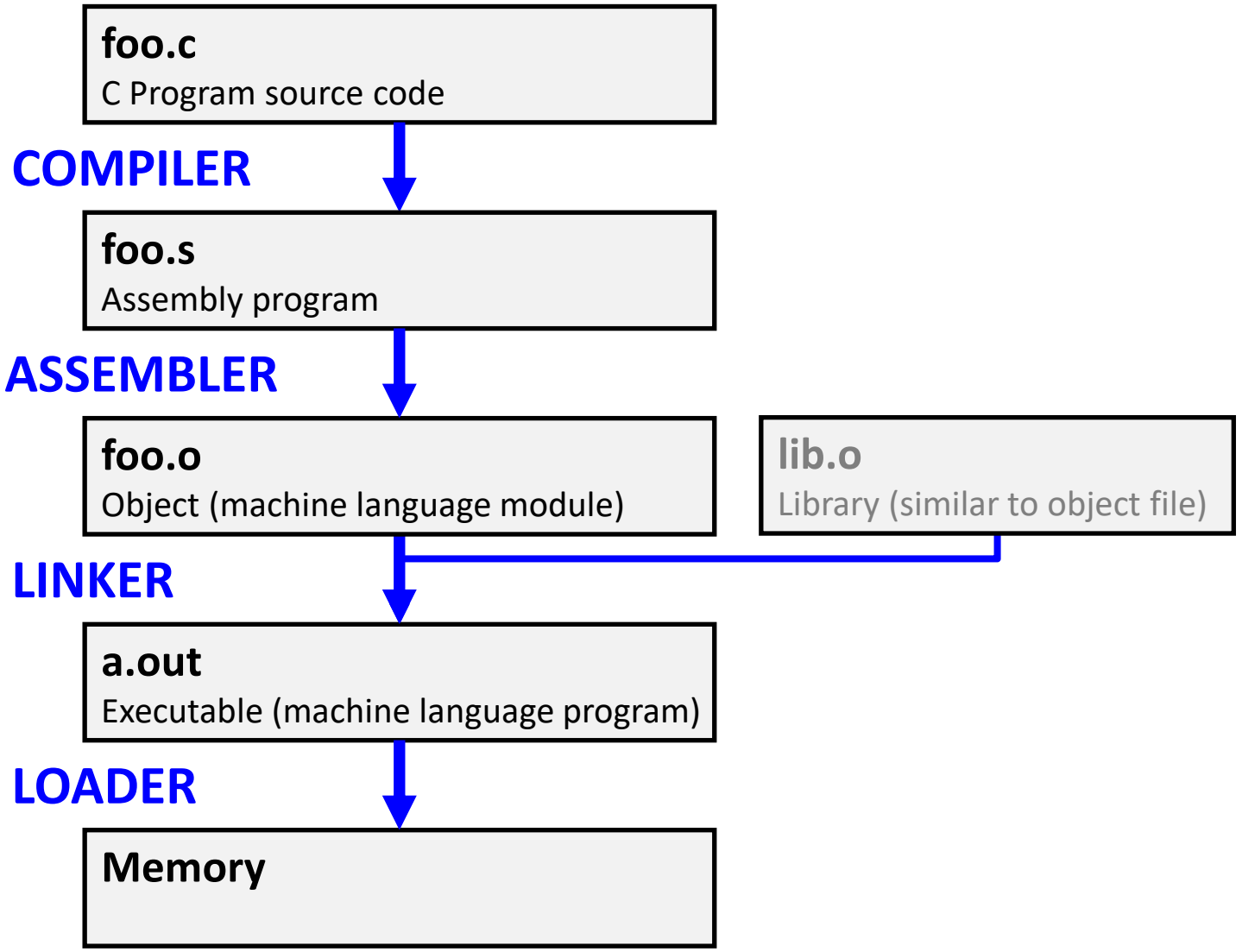
Compiler

- Input: High-Level Language Code (e.g., C)
- Output: Assembly Language Code (e.g., MIPS)
  - In MARS we use **.asm** as a file extension (default), but **.s** is a common file extension for assembly in other scenarios and can execute on MARs as well.
- Note: Output *may* contain pseudoinstructions
  - Assembler understands these instructions, but not the machine

Compiler and Standards

- Compiler generates assembly code and directives that respect conventions
  - For example, function call register conventions
  - There are many more details concerning data representation and function linkage which are *beyond the scope of this course*

Steps to Starting a Program



Assembler

- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

Assembler Directives (B.2, B.9, B.10)

- Directives provide directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text (instructions) segment
  - `.data`: Subsequent items put in user data segment
  - `.globl sym`: declares **`sym`** global allowing reference from other files
  - `.asciiz str`: Store string **`str`** in memory and null-terminate it
  - `.word w1…wn`: Store *n* 32-bit words in successive memory locations

Pseudoinstruction Replacement

Assembler treats convenient variations of machine language instructions as if real (see B.10)

Pseudo (MAL):

```
addu $t0,$t6,1
sd   $a0,32($sp)

ble  $t0,100,loop
```

Real (TAL):

```
addiu $t0,$t6,1
sw    $a0,32($sp)
sw    $a1,36($sp)
slti  $at,$t0,101
bne   $at,$0,loop
```

Producing Machine Language (1/2)

- Simple instructions for Assembler
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- What about Branches?
  - PC-Relative
  - Once pseudoinstructions are replaced by real ones, we know by how many instructions to branch
- So these 2 cases are handled easily

Producing Machine Language (2/2)

- What about jumps (**`j`** and **`jal`**)?
  - Jumps require **absolute address**
- What about references to data?
  - **Ex:** `la, lw, sw`
  - These will require the full 32-bit address of the data
- These can't be determined yet
  - Must wait to see where this code will appear in final program
- Two tables are used to help assembly and later resolution of addresses

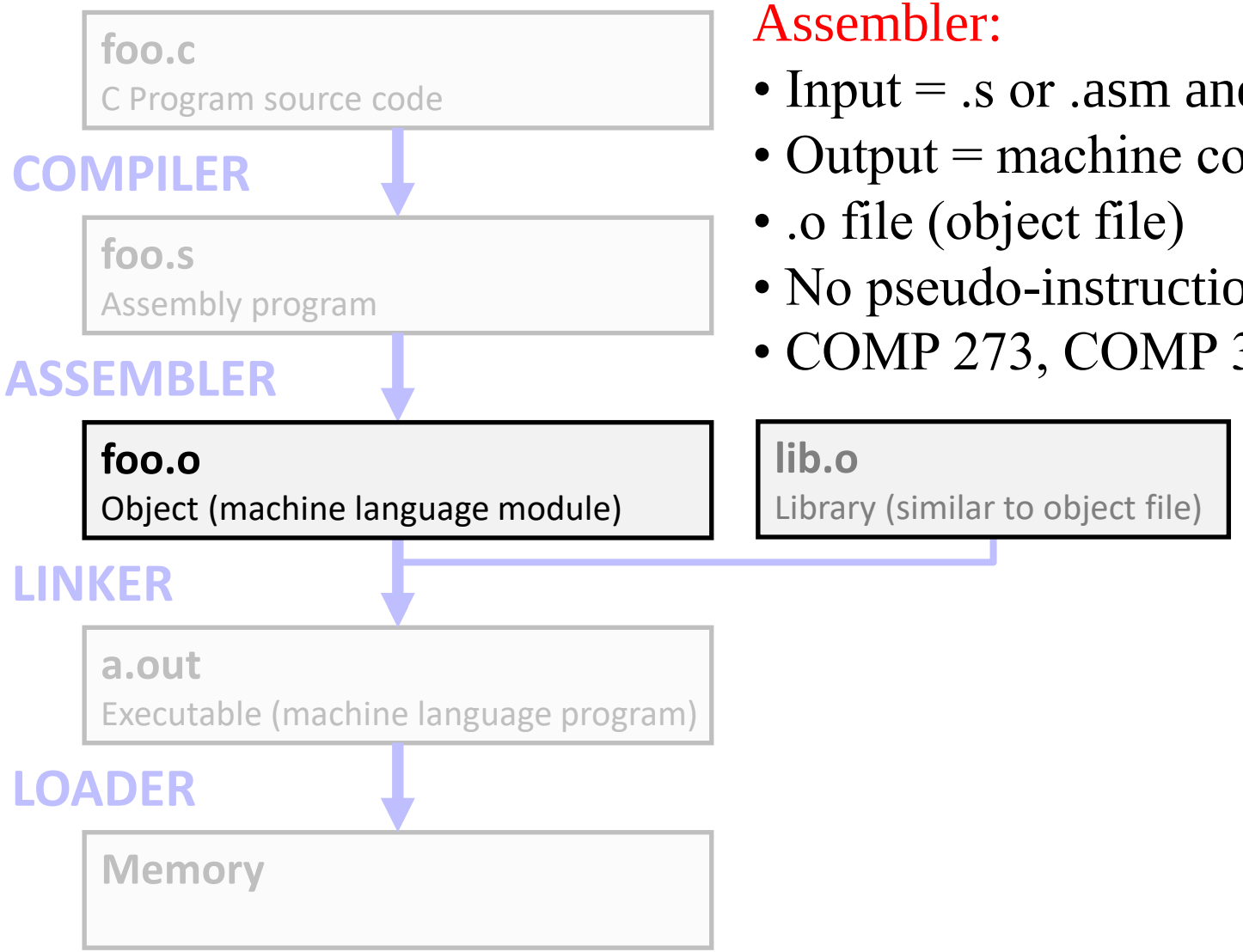
1st Table: Symbol Table

- Symbol table: List of "items" in this file that may be used by this and other files
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables which may be accessed across files
- First Pass: record label-address pairs
- Second Pass: produce machine code
  - Result: can jump to a label later in code without first declaring it

2nd Table: Relocation Table

- Relocation Table: line numbers of “items” in this file which need the address filled in (or fixed up) later.
- What are they?
  - Any reference to code that is not part of the local file.
    - Ex. Printf().
  - References to things that do not exist in your code are flagged for further analysis.

Steps to Starting a Program



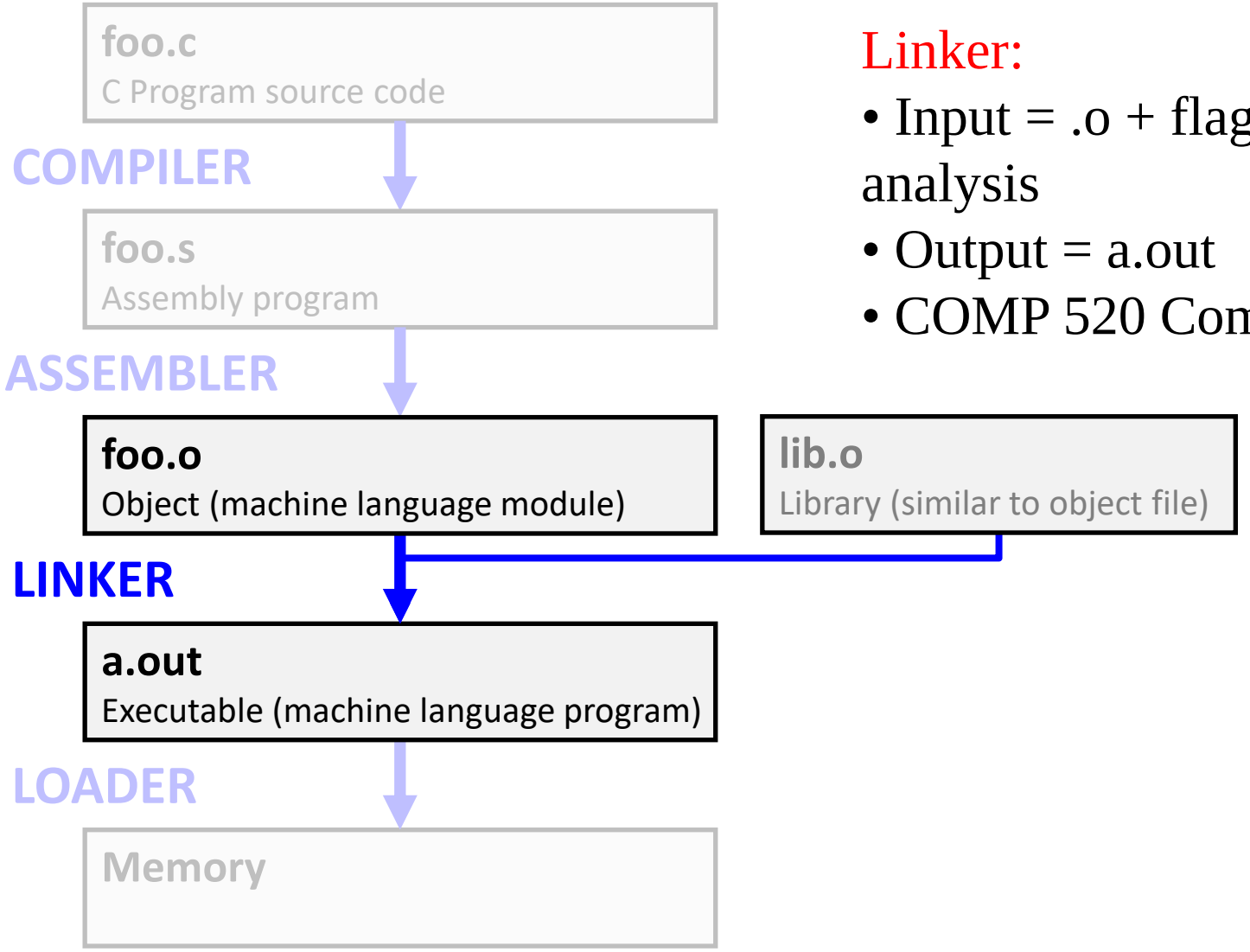
Assembler:

- Input = .s or .asm and directives
- Output = machine code
- .o file (object file)
- No pseudo-instructions
- COMP 273, COMP 308

Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation table: identifies lines of code that need to be “handled”
- symbol table: list of this file’s labels and data that can be referenced
- debugging information

Steps to Starting a Program



Linker:

- Input = .o + flagged for analysis
- Output = a.out
- COMP 520 Compiler Design

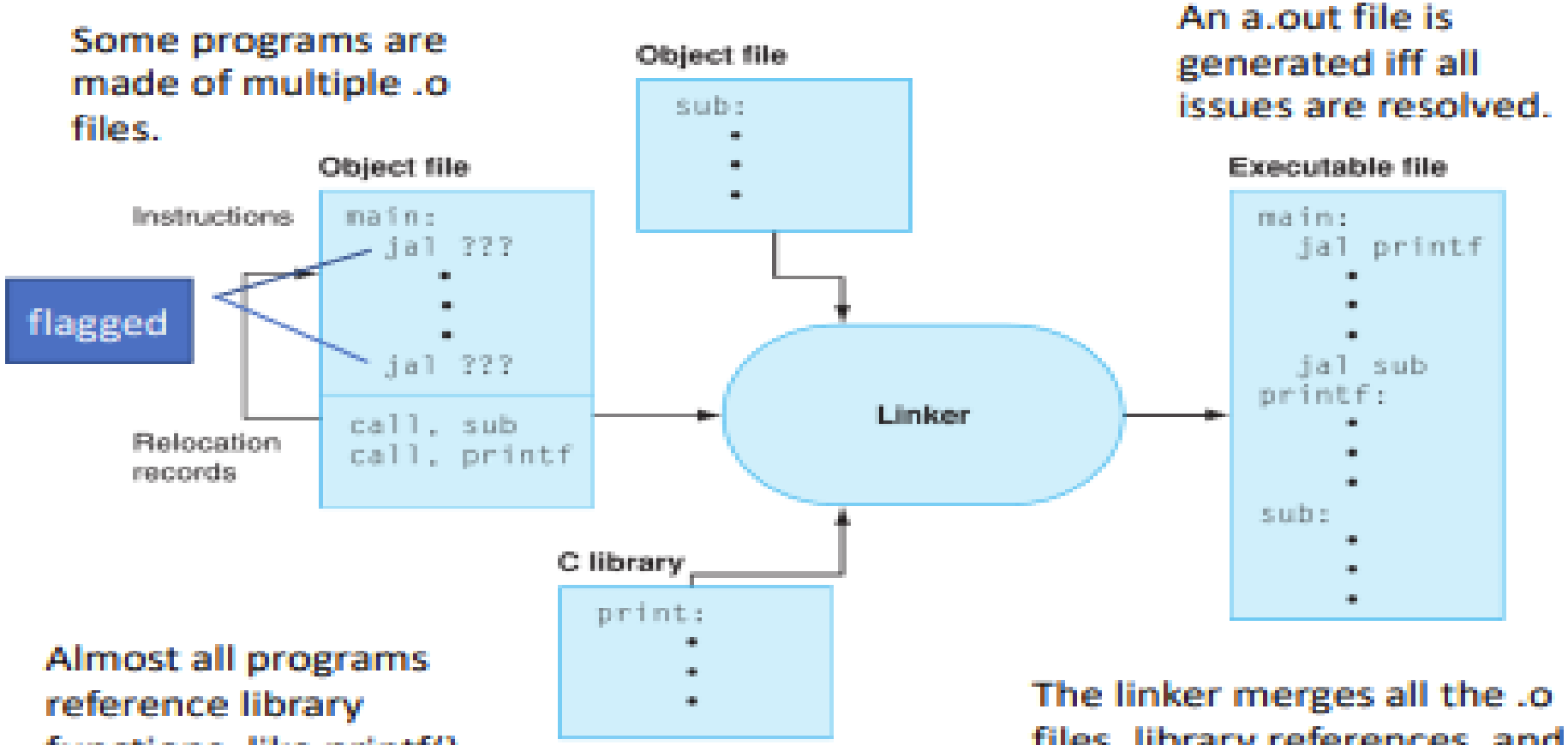
Linker (Link Editor) (1/3)

- What does Link Editor do?
- Combines several object (.o) files into a single executable ("linking")
- Enables *Separate Compilation* of files
  - Changes to one file do not require recompilation of whole program
    - Linux kernel source: > 6 M lines of code
    - Windows OS source: > 40 M lines of code
  - Code in file called a module
  - Link Editor name from editing the "links" in jump and link instructions

Link Editor/Linker (2/3)

- Step 1: Combine text segment from each .o file
- Step 2: Combine data segment from each .o file, and concatenate this onto end of text segments
- Step 3: Resolve References
  - Go through Relocation Table
  - Handle each entry using the Symbol Table
    - That is, fill in all absolute addresses

Link Editor/Linker (3/3)



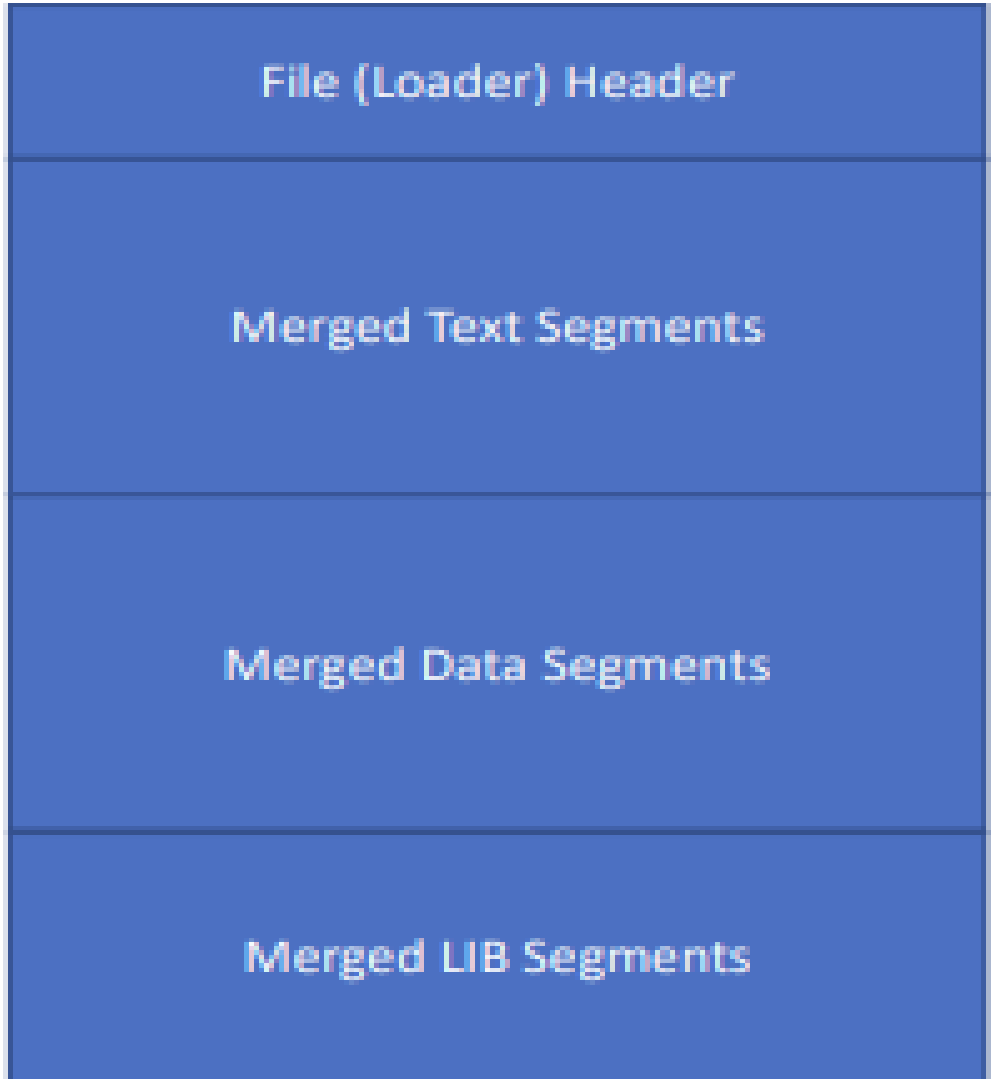
Resolving References (1/2)

- Linker *assumes* first word of first text segment is at address 0x00000000
- Linker knows:
  - Length of each text and data segment
  - Ordering of text and data segments
- Linker calculates:
  - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced from zero.

Resolving References (2/2)

- To resolve references:
  - Search for reference (data or label) in all symbol tables
  - If not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker:
  - Executable file containing text and data (plus a file header): a.out

Format of a.out file



The File Header contains information like: max run-time stack size needed, max heap size needed, pointer to first instruction, requests to OS for other resources.

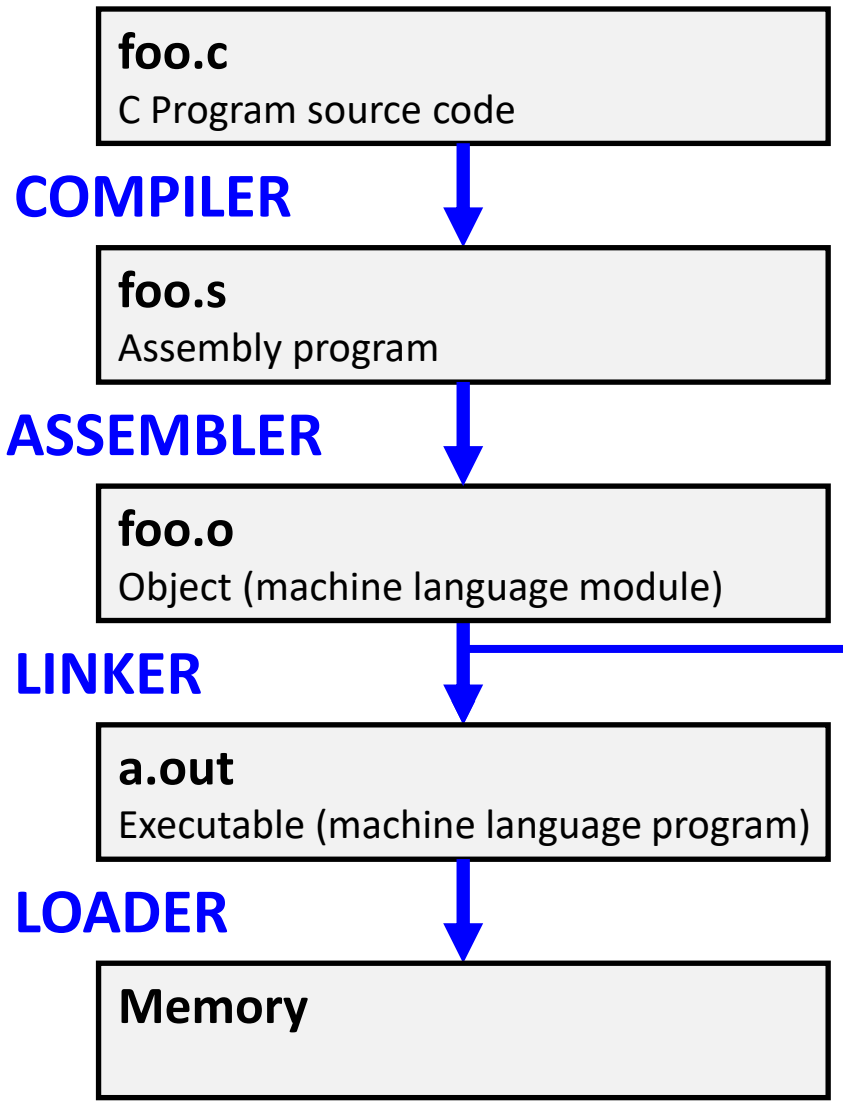
Libraries exist in two formsL LIB and DLL. LIB (Library) are appended to the a.out file. DLL (Dynamic Link Libraries) are appended to the OS and accessible with an external function call. (.so in Unix, Shared Object)

DLL Segments (OS managed)

Question

- Are both .O and .Out files having machine code? If yes, then what is the difference between them?

Steps to Starting a Program



Loader:

- Input = a.out
- Output = RAM & CPU
- Program copied to RAM
- Run-time stack created
- Heap created
- Pointer to first instruction given to CPU

Loader (1/3)

- Executable files are stored on disk.
- When one is to be run, loader's job is to load it into memory and start it running.
- In reality, loader is the operating system (OS)
  - Loading is one of the OS tasks

Loader (2/3)

- So what does a loader do?
- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space

Loader (3/3)

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer must be initialized to top of the stack memory space
- Jumps to start-up routine that copies program's arguments from stack to registers and sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Dynamic Linking

- Some operating systems allow “dynamic linking”
- Both the loader *and* the linker are part of the operating system - so modules can be linked and loaded at runtime
- If a module is needed and already loaded, it need not be loaded again
- Called DLLs in Windows, `.so` in Unix
  (Dynamically Linked Library / Shared Object)

C → Asm → Obj → Exe → Run

Compile C Source

Let us consider compilation of the following code…

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        int i;
        int prod = 0;
        for (i = 0; i <= 100; i = i + 1) {
          prod = prod + i * i;
        }
        printf ("The sum squares from 0 .. 100 is %d\n", prod);
}
```

C → **Asm** → Obj → Exe → Run

Identify Pseudoinstructions

```
.text
   .align     2
   .globl     main
main:
   subu $sp,$sp,32
   sw   $ra, 20($sp)
   sd   $a0, 32($sp)
   sw   $0, 24($sp)
   sw   $0, 28($sp)
loop:
   lw   $t6, 28($sp)
   mul $t7, $t6,$t6
   lw   $t8, 24($sp)
   addu $t9,$t8,$t7
   sw   $t9, 24($sp)
   addu $t0, $t6, 1
   sw   $t0, 28($sp)
   ble $t0,100, loop
   la   $a0, str
   lw   $a1, 24($sp)
   jal printf
   move $v0, $0
   lw   $ra, 20($sp)
   addiu $sp,$sp,32
   j    $ra
.data
.align 0
str:
.asciiz       "The product
   from 0 .. 100 is %d\n"
```

FINE PRINT: The modification of the stack pointer may look strange, but this is ultimately from a *real example of compilation*… a number of the real details are being omitted here, some of which we will see later.

C → Asm → **Obj** → Exe → Run

Remove Pseudoinstructions, Assign Addresses

```
00 addiu $29,$29,-32
04 sw  $31,20($29)
08 sw  $4, 32($29)
0c sw  $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw   $14, 28($29)
1c mult $14, $14
20 mflo       $15
24 lw   $24, 24($29)
28 addu $25,$24,$15
2c sw   $25, 24($29)
30 addiu $8,$14, 1
34 sw  $8,28($29)
38 slti       $1,$8, 101
3c bne $1,$0, loop
```

```
40 lui $4, l.str
44 ori $4,$4, r.str
48 lw  $5,24($29)
4c jal printf
50 addu       $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr   $31
```

C → Asm → **Obj** → Exe → Run

Symbol Table Entries

- Symbol Table

Label	Address
`main:`	`0x00000000`
`loop:`	`0x00000018`
`str:`	`0x10000430`
`printf:`	`-`

- Relocation Table

Address	Instruction/Type	Dependency
`0x0000004c`	`jal`	`printf`

C → Asm → **Obj** → Exe → Run

Edit Local Addresses

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25,$24,$15
2c sw    $25, 24($29)
30 addiu $8,$14, 1
34 sw    $8,28($29)
```

```
38 slti  $1,$8, 101
3c bne   $1,$0, -10
40 lui   $4, 0x1000
44 ori   $4,$4,0x0430
48 lw    $5,24($29)
4c jal   0
50 addu  $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr    $31
```

Can fix several of these labels now, while others (0x4c) are left for later

C → Asm → **Obj** → Exe → Run

```
0x000000  00100111101111011111111111100000
0x000004  10101111101111110000000000010100
0x000008  10101111101001000000000000100000
0x00000c  10101111101001010000000000100100
0x000010  10101111101000000000000000011000
0x000014  10101111101000000000000000011100
0x000018  10001111101011100000000000011100
0x00001c  00000001110011100000000000011001
0x000020  00000000000000000111100000010010
0x000024  10001111101110000000000000011000
0x000028  00000011000011111100100000100001
0x00002c  10101111101010000000000000011100
0x000030  00100101110010000000000000000001
0x000034  10101111101110010000000000011000
0x000038  00101001000000010000000001100101
0x00003c  00010100001000001111111111110111
0x000040  00111100000001000001000000000000
0x000044  00110100100001000000010000110000
0x000048  10001111101001010000000000011000
0x00004c  00001100000100000000000011101100
0x000050  00000000000000000001000000100001
0x000054  10001111101111110000000000010100
0x000058  00100111101111010000000000100000
0x00005c  00000011111000000000000000001000
```

C → Asm → Obj → **Exe** → Run

- Combine with object file containing “printf”
- Edit absolute addresses
  - In this case edit **`jal printf`** to contain actual address of printf
- Output single binary file

hello.c with gcc on Window 10

```
/* hello.c */
#include <stdio.h>
int main( int argc, char** argv ) {
    printf("Hello COMP273");
}
```

gcc –S hello.c --> hello.s

hello.s X86 Assembly

```
            .file       "hello.c"
            .def        ___main;    .scl        2;          .type       32;         .endef
            .section .rdata,"dr"
LC0:
            .ascii "Hello COMP273\0"
            .text
            .globl      _main
            .def        _main;      .scl        2;          .type       32;         .endef
_main:
LFB13:
            .cfi_startproc
            pushl       %ebp
            .cfi_def_cfa_offset 8
            .cfi_offset 5, -8
            movl        %esp, %ebp
            .cfi_def_cfa_register 5
            andl        $-16, %esp
            subl        $16, %esp
            call        ___main
            movl        $LC0, (%esp)
            call        _printf
            movl        $0, %eax
            leave
            .cfi_restore 5
            .cfi_def_cfa 4, 4
            ret
            .cfi_endproc
LFE13:
            .ident      "GCC: (Rev3, Built by MSYS2 project) 5.2.0"
            .def        _printf;    .scl        2;          .type       32;         .endef
```

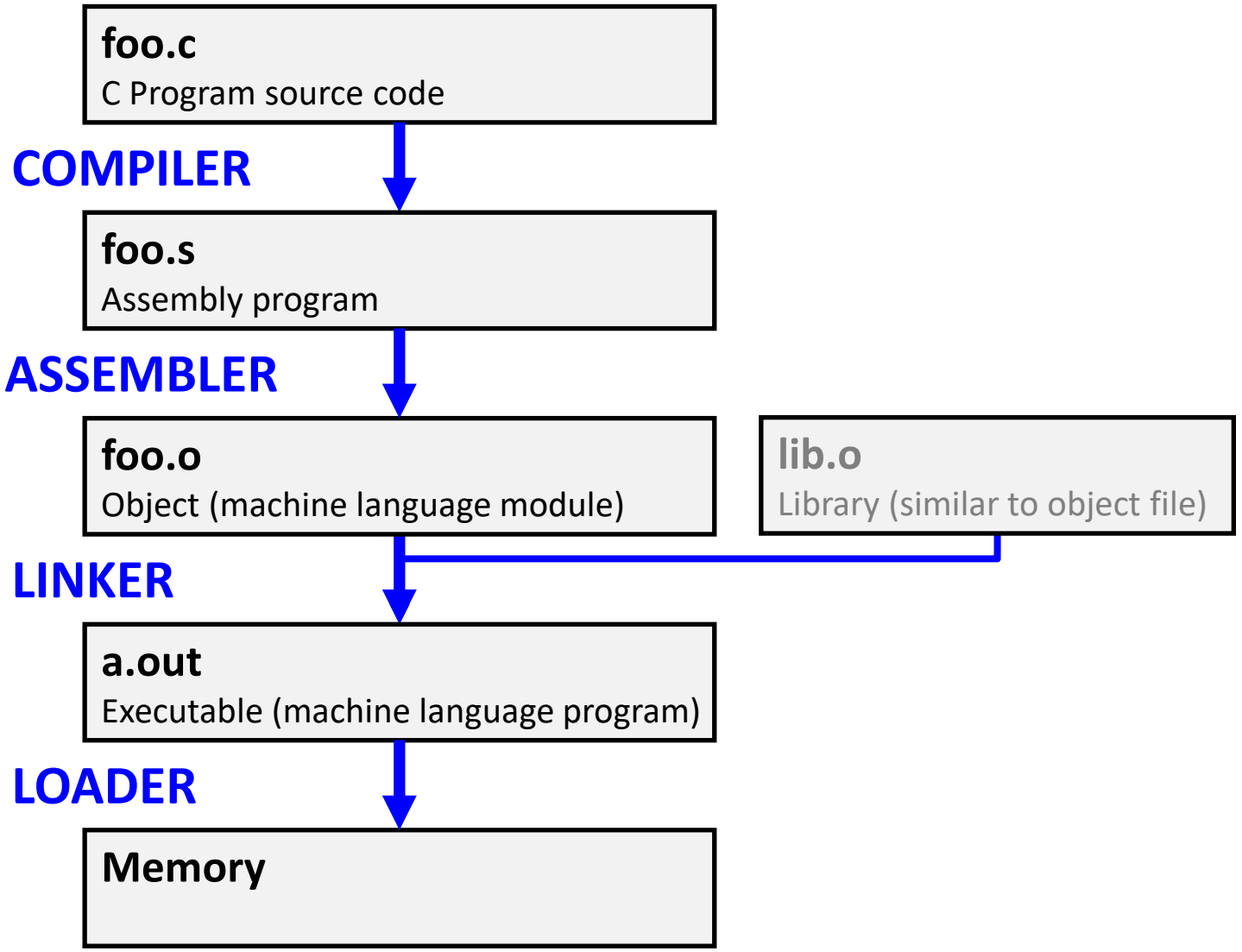
Things to Remember 1/3

- Compiler → Assembler → Linker (→ Loader)
- Assembler does ***2 passes*** to resolve addresses, handling internal forward references
- Linker enables ***separate compilation***, libraries that need not be compiled, and resolves remaining addresses

Things to Remember (2/3)

- Compiler converts a single HLL file into a single assembly language file
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file
- Linker combines several .o files and resolves absolute addresses
- Loader loads executable into memory and begins execution

Steps to Starting a Program



Review and More Information

- Textbook 5th edition, A.2 and A.3
  - (B2 and B3 of 4th edition)
- Chapter 2 Section 12, translating and starting your program.