ECE260: Fundamentals of Computer Engineering

Translation of High-Level Languages

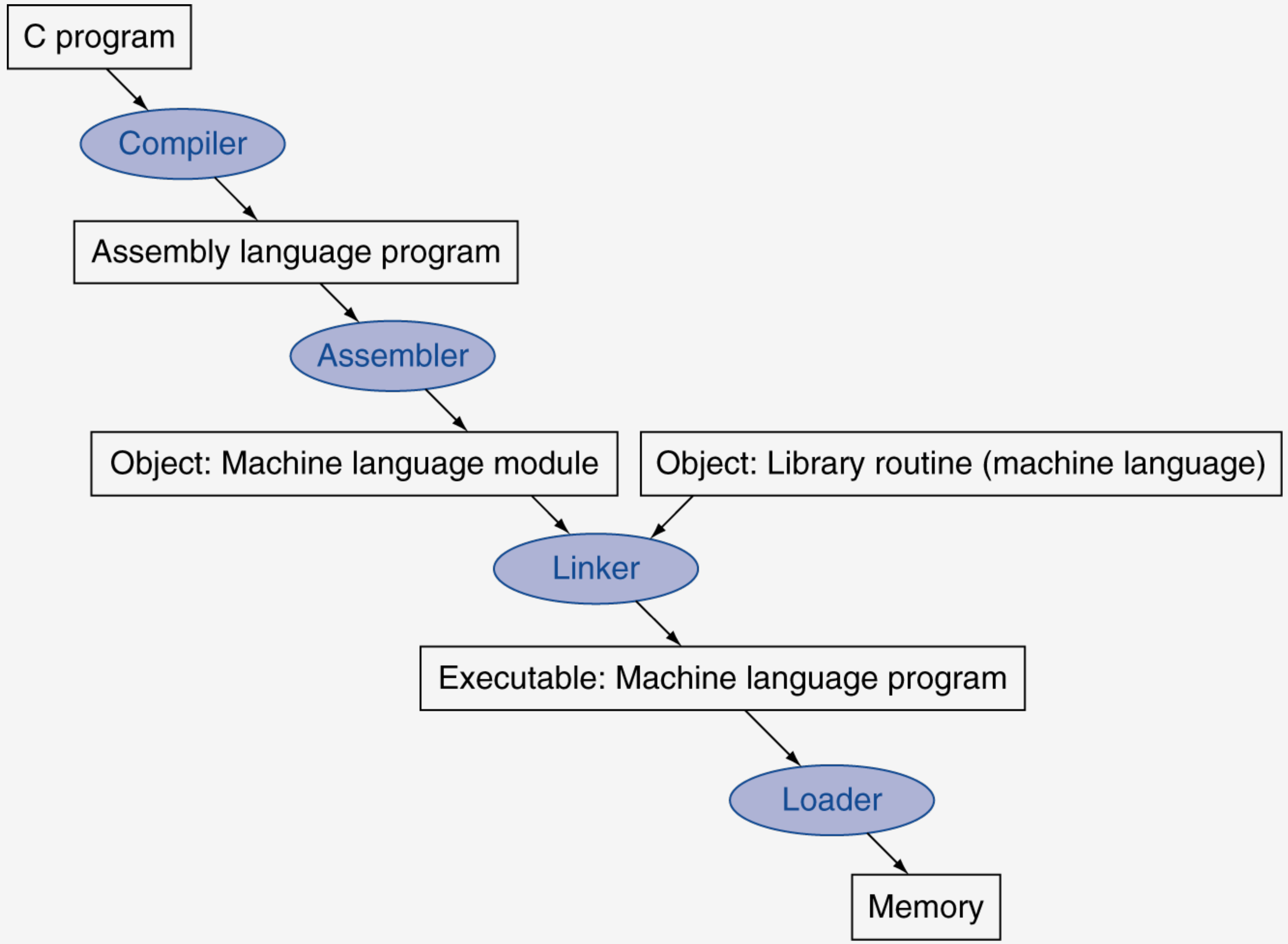
James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

Translation of High-Level Languages

- Writing code in assembly is time consuming and can be challenging
  - A single line of C code may require many lines of assembly
    - Example: D[4 + i] = A[5 * j] + 6;
  - Must manage limited register set and stack
- High-level languages exist to make programming computers easier
  - Abstract away many of the complexities of the underlying hardware
  - Increase programmer productivity
- ***Compilers*** exist to automate translation from a high-level language into assembly
  - Typically integrated with an ***assembler*** and a ***linker*** to produce executable code

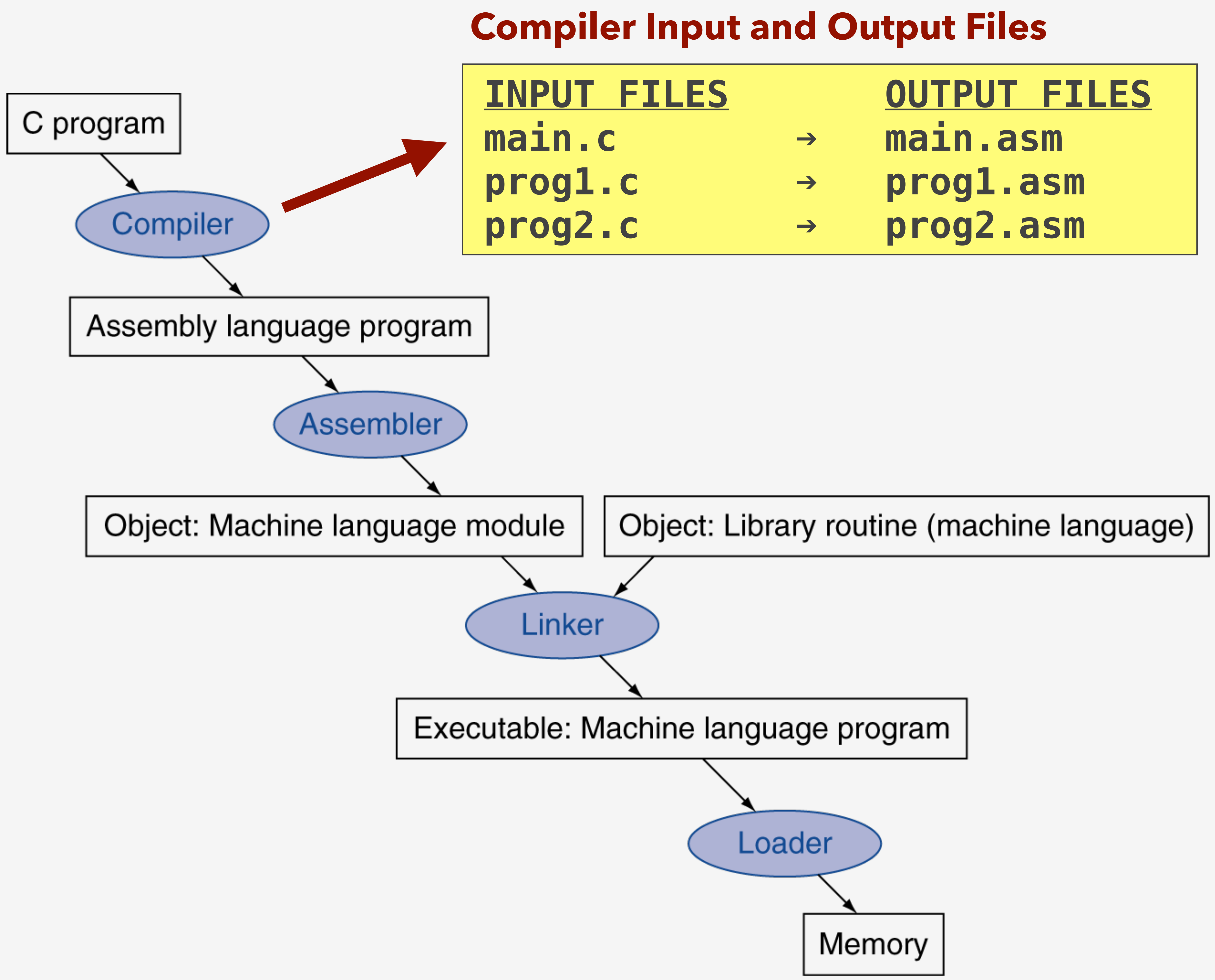
Translation and Startup of a Program

- Translation and startup of a program includes the following steps:
  - Programmer writes some code
  - Compiler translates code into assembly
  - Assembly is converted into an object file
  - Linker “stitches” together object files to produce an executable file
- At a later time, when the executable file is executed
  - A loader loads the executable instructions and data into memory



Compiling Code

- Programmer may write code in multiple files
  - Program written in C may include many ".c" files
- For each file, a ***compiler*** parses and translates C code into assembly code
  - Basic blocks of code are converted into assembly
  - Conditions are translated into branch and jump instructions
  - Procedures are translated using conventions for saving/restoring registers and manipulating the stack
  - May perform variety of optimizations



Assembling Into Object Files

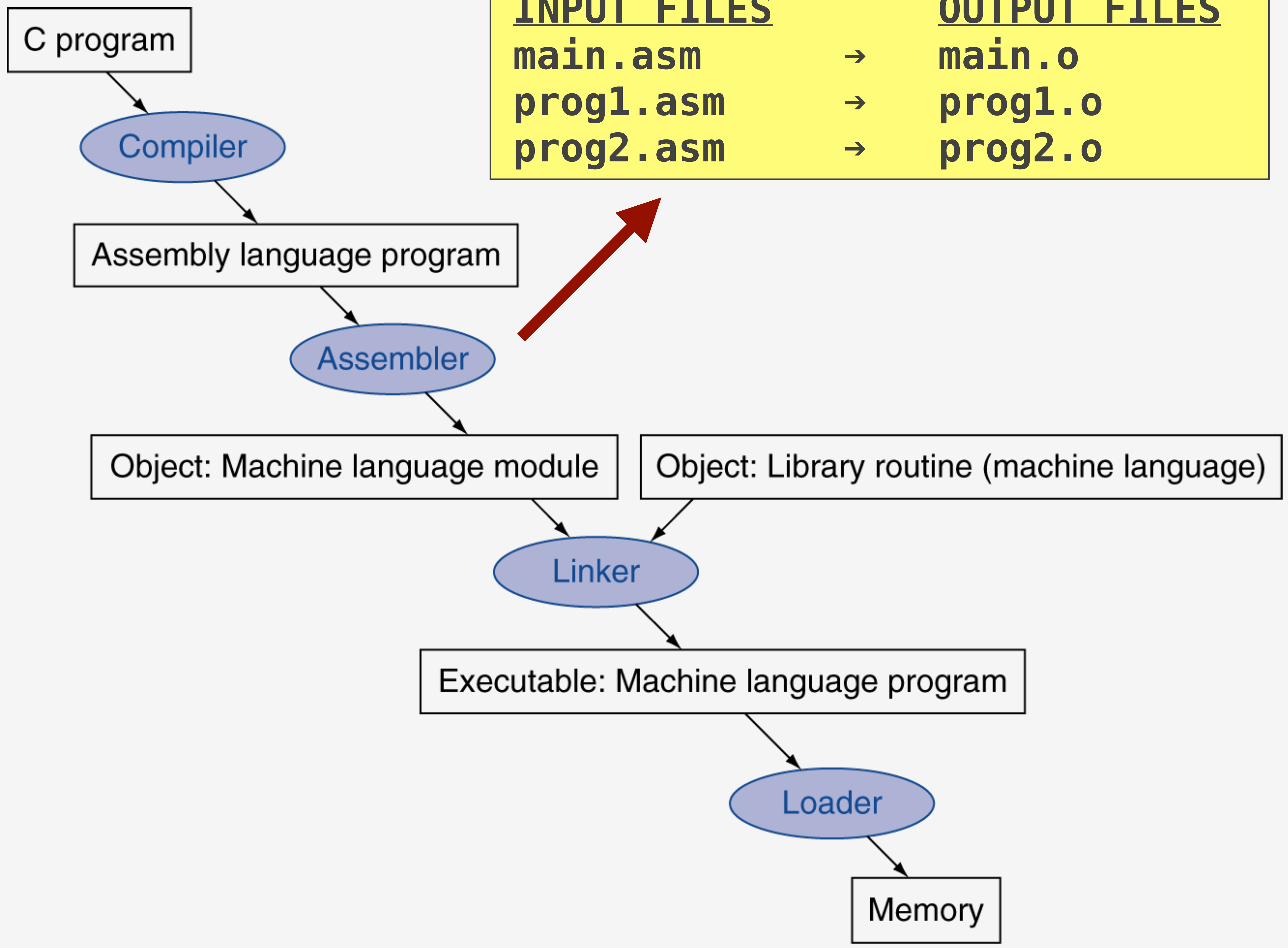
- For each input file, an ***assembler*** converts assembly into object files (e.g. ".o" files)
  - Native instructions in the input .asm files are converted directly into machine code
  - Pseudoinstructions (e.g. move, bgt, ble, etc.) are converted into native instructions which are converted into machine code

```
move $t0, $t1  →  add $t0, $zero, $t1
```

  - A ***symbol table*** is created in object file
    - Label name/memory address mapping
    - Memory addresses are relative to beginning of each object file (this will change later)

Assembler Input and Output Files

INPUT FILES		OUTPUT FILES
main.asm	→	main.o
prog1.asm	→	prog1.o
prog2.asm	→	prog2.o

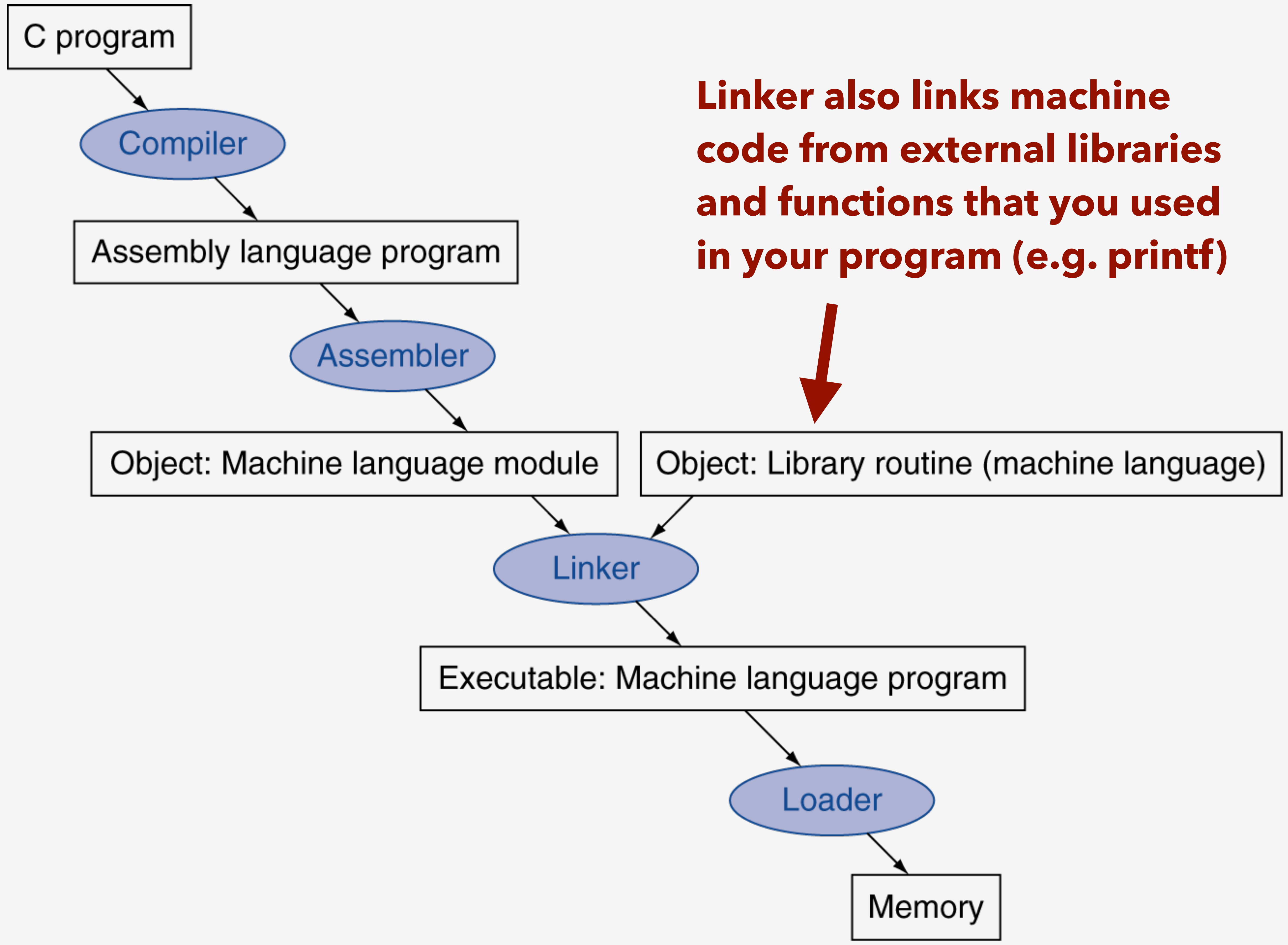


Anatomy of an Object File

- Object file provides information for building a complete program from the multiple input files
  - ***Object File Header*** – describes the size and position of the other pieces of the object file
  - ***Text Segment*** – contains the translated assembly instructions as machine code
  - ***Static Data Segment*** – contains global and static data that is allocated for the life of the program
  - ***Relocation Info*** – contains location of instructions and data words that may need modification when multiple object files are linked together
  - ***Symbol Table*** – contains list of labels that are not defined in the input file (i.e. external references)
    - These will be resolved later when multiple object files are linked together
  - ***Debug Info*** – contains info for associating machine code with original high-level language code
    - Enables your IDE to step through code!

Linking Object Files

- A ***linker*** combines all of the object files that constitute a program into a single ***executable file***
  - Merges .text and .data segments from multiple object files into single .text and .data segments
  - Determines addresses of data and instruction labels in newly combined file
  - Patches memory addresses that were relative to individual object files – now relative to combined object file
- This approach reduces compile time since only high-level language files that have changed need to be recompiled



Linker also links machine code from external libraries and functions that you used in your program (e.g. printf)

Linking Object Files (continued)

Object File #1

OBJECT FILE HEADER
Text Size: 100_{hex}
Data Size: 20_{hex}

TEXT SEGMENT

DATA SEGMENT

RELOCATION INFORMATION

SYMBOL TABLE

Object File #2

OBJECT FILE HEADER
Text Size: 200_{hex}
Data Size: 30_{hex}

TEXT SEGMENT

DATA SEGMENT

RELOCATION INFORMATION

SYMBOL TABLE

Object files are merged into a single executable file

Executable File

OBJECT FILE HEADER
Text Size: 300_{hex}
Data Size: 50_{hex}

TEXT SEGMENT (from Obj #1)

TEXT SEGMENT (from Obj #2)

DATA SEGMENT (from Obj #1)

DATA SEGMENT (from Obj #2)

Linking Object Files – Example

Object File #1

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	

Object File #2

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Linked Object Files (i.e. executable file)

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	`lw $a0, 8000hex($gp)`
	$0040\ 0004_{hex}$	`jal 40 0100hex`
	...	...
	$0040\ 0100_{hex}$	`sw $a1, 8020hex($gp)`
	$0040\ 0104_{hex}$	`jal 40 0000hex`
	...	...
Data segment	Address	
	$1001\ 0020_{hex}$	(X)
	...	...
	$1001\ 0020_{hex}$	(Y)
	...	...

Assume:

- **.text segment starts at $0040\ 0000_{hex}$**
- **.data segment starts at $1001\ 0000_{hex}$**
- **\$gp is initialized to $1000\ 8000_{hex}$**

Loading a Program

- ***Loader*** loads executable file from disk into memory
  - Reads header to determine size of text and data segments
  - Creates an address space large enough for the text and data
  - Copies the instruction and data from the executable file into memory
  - Copies parameters (if any) to the main program onto the stack (e.g. command line arguments)
  - Initializes machine registers (including $sp, $fp, $gp)
  - Jumps to a startup routine that copies parameters into argument registers (i.e. $aX) and calls “main”
  - When “main” returns, the startup routine terminates the program

Static vs. Dynamic Linking

- ***Static linking*** occurs during the linking phase of the compilation process
  - Objects that are statically linked are combined into the executable output file
  - All libraries that are used compile directly into executable
    - Increases executable file size
    - Library updates require generation of a new executable file :-(
- ***Dynamic linking*** occurs at runtime and only links/loads code when it is called
  - Often used when linking with operating system libraries (e.g. Windows .DLL files)
  - Avoids executable file bloat
  - Automatically links with updated libraries next time program is run
    - NOT ALWAYS A GOOD THING