From Source to Execution: Translation and Linking

CSE 410, Spring 2009

Computer Systems

http://www.cs.washington.edu/410

Readings and References

- Reading
  - » Section 2.12, Translating and Starting a Program
  - » Appendix B.1, Introduction
  - » Appendix B.2, Assemblers
  - » Appendix B.3, Linkers
  - » Appendix B.4, Loading

Starting a Program

- Two phases from source code to execution
- Build time
  - » compiler creates assembly code
  - » assembler creates machine code
  - » linker creates an executable

  (Spim assembles/links when file loaded)
- Run time
  - » loader moves the executable into memory and starts the program

Build Time

- You’re experts on generating assembly language: either by writing high-level code that is compiled, or by hand
- Two parts to translating from assembly to machine language:
  - » Instruction encoding (including translating pseudoinstructions)
  - » Translating labels to addresses
- Label translations go in the *symbol table*
  - » Symbol table: map from labels (names) to their addresses in the code

Modular Program Design

- Small projects might use only one file
  - » Any time any one line changes, recompile and reassemble the whole thing
- For larger projects, recompilation time and complexity management is significant
- Solution: split project into modules
  - » compile and assemble modules separately
  - » link the object files

The Compiler + Assembler

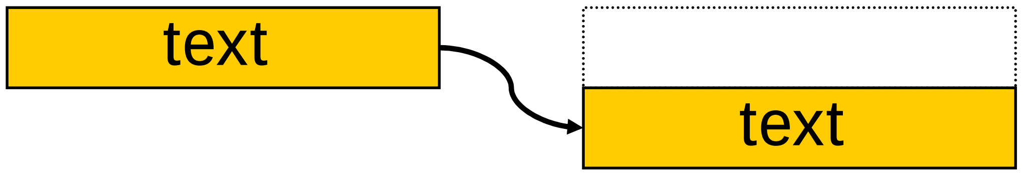
- Translate source files to object files
- Object files
  - » Contain machine instructions (1’s & 0’s)
  - » Contain bookkeeping information
    - Procedures and variables the object file defines (globals)
    - Procedures and variables the object file uses but does not define (unresolved [or external] references)
    - Debugging information associating machine instructions with lines of source code

The Linker

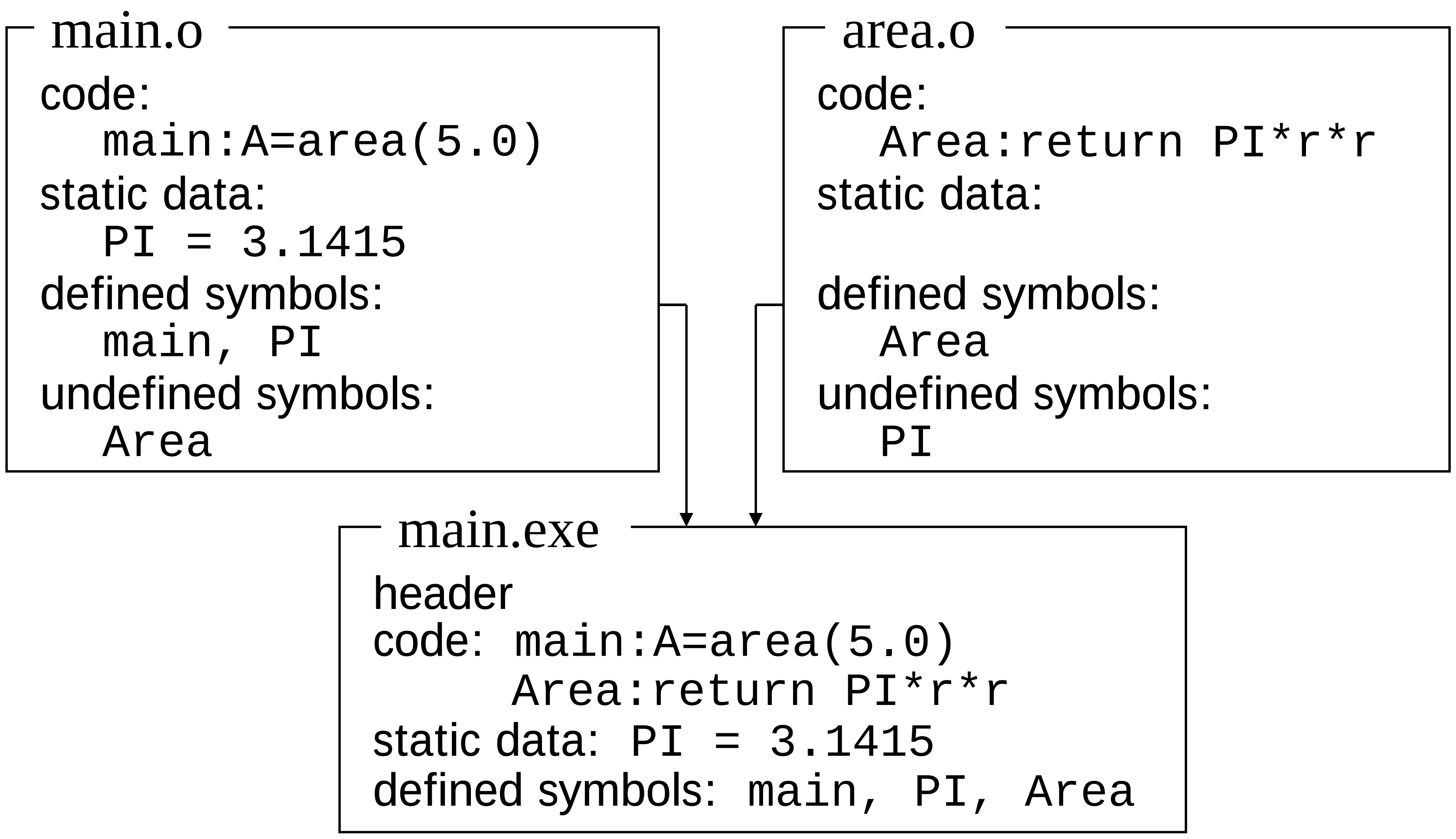
- The linker’s job is to “stitch together” the object files:
  1. Place the modules in memory space
  2. Determine the addresses of data and labels
  3. Match up references between modules
- Creates an executable file

Determining Addresses

- Some addresses change during memory layout
- Modules were compiled/assembled in isolation
  - » Assembler assigns addresses starting at 0 during assembly
  - » Final addresses assigned by linker
- *Absolute* addresses must be *relocated*
- Object file keeps track of instructions that use absolute addresses



Linker Example



main.o

```
code:
   main:A=area(5.0)
static data:
   PI = 3.1415
defined symbols:
   main, PI
undefined symbols:
   Area
```

area.o

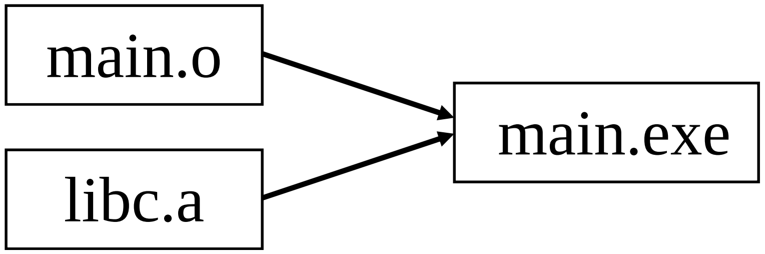
```
code:
   Area:return PI*r*r
static data:

defined symbols:
   Area
undefined symbols:
   PI
```

main.exe

```
header
code:  main:A=area(5.0)
       Area:return PI*r*r
static data:  PI = 3.1415
defined symbols:  main, PI, Area
```

Libraries

- Some code is used so often, it is bundled into *libraries* for common access
- Libraries contain most of the code you use but didn't write: e.g., printf(), sqrt()
- Library code is (often) merged with yours at link time



The Executable

- End result of compiling, assembling, and linking: the *executable*
  - » Header, listing the lengths of the other segments
  - » Text (code) segment
  - » Static data segment
  - » Potentially other segments, depending on architecture & OS conventions

Run Time

- When a program is started ...
  - » Some *dynamic linking* may occur
    - some symbols aren’t defined until run time
    - Windows’ dlls (dynamic link library)
  - » The segments are loaded into memory
  - » The OS transfers control to the program and it runs
- We’ll learn a lot more about this during the OS part of the course