


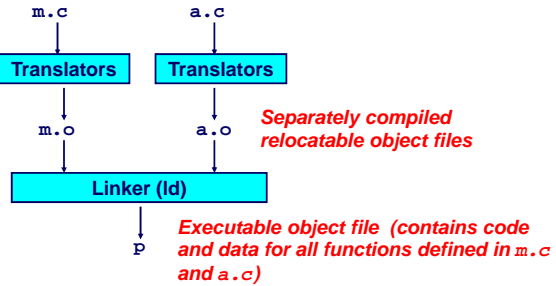
# Linking

CSE 361S

## One Step Translation

- Simple translation will work:
 
- Problems:
  - Efficiency: small change requires complete recompilation
  - Modularity: hard to share common functions (e.g. printf)
- Solution:
  - Static linker (or linker)

## Linker



## Linker (ld)

- Merges multiple relocatable (.o) object files into a single executable object file
- Resolves external references
  - External reference: reference to a symbol defined in another object file.
- Relocates symbols from their relative locations in the .o files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions. References can be in either code or data
  - code: a(); /\* reference to symbol a \*/
  - data: int \*xp=&x; /\* reference to symbol x \*/

## Executable and Linkable Format (ELF)

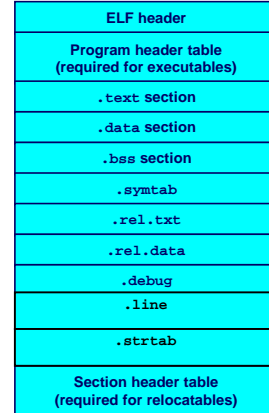
- Standard format for many Unix-style operating systems. It handles:
  - relocatable object files (.o)
  - executable binaries
  - shared object files (.so)

## ELF file contents (1)

- header: magic number, file type (relocatable, binary, shared), machine type, machine properties (word size, byte order)
- program header table: page size, virtual memory addresses, segments (sections), segment sizes
- .text section: code
- .data section: initialized data (static data only)
- .bss section: uninitialized data (static data only)

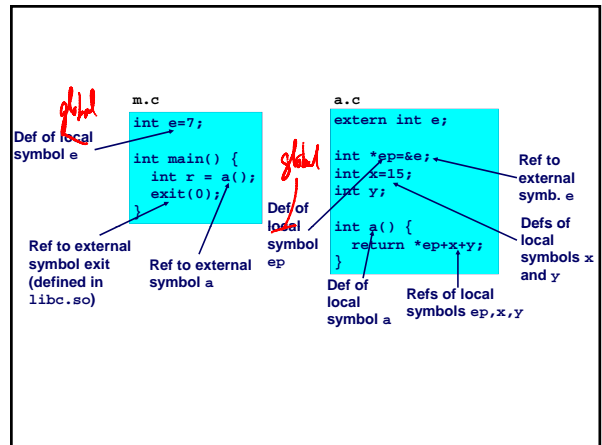
### ELF file contents (2)

- .symtab section: global symbols
- .rel.text section: relocation information for code
- .rel.data section: relocation information for data
- .debug section: debugging symbol table (-g)
- .line section: mapping line numbers to machine code (-g)
- .strtab section: constant strings table
- section header table: where is above stuff located in ELF file



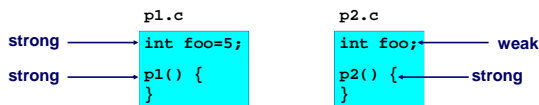
### Old Names / New Definitions

- A *module* is a file
- *Global* symbols – defined in a module and can be referenced by other modules
  - Functions and variables w/o static attribute
- *External* symbols – defined in a remote module and referenced in the current module
- *Local* symbols – defined and referenced only within the current module
  - Functions and variables declared static



### Symbol Strength

- Program symbols are either “strong” or “weak”
  - strong: procedure names and initialized global variables
  - weak: uninitialized global variables



### Linking Rules

- Rule 1. A strong symbol can only appear once.
- Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
  - References to the weak symbol resolve to the strong symbol.
- Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

```
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!

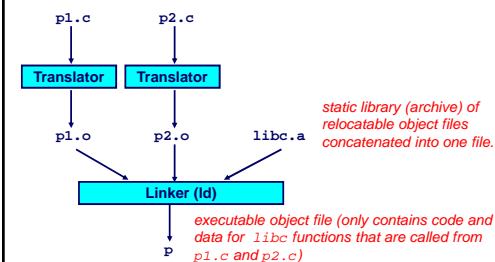
```
int x=7;
int y=7;
p1() {}    double x;
p2() {}
```

- Writes to x in p2 will overwrite y!
- Nasty!

```
int x=7;
p1() {}    int x;
p2() {}
```

- References to x will refer to the same initialized int.

**Nightmare scenario:** two identical weak structs, compiled by different compilers with different alignment rules.



- Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (libc), math library (libm)].
- Linker selectively links only the .o files in the archive that are actually needed by the program.

