Quick notes

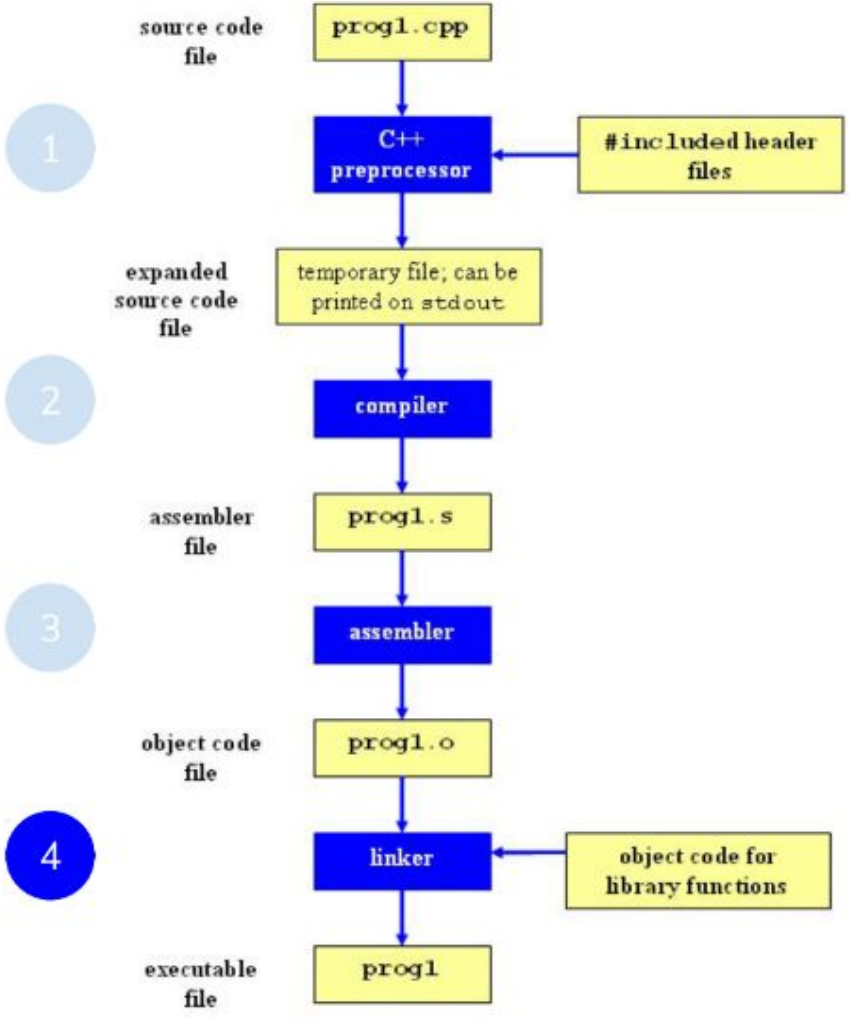
- Check Piazza
- Reminders:
  - Assign6 is due friday 20th Nov
  - Assign7 due 29th Nov
  - Weekly quiz
- Assignment 10 (<u>this only applies to students enrolled in Lab 1</u>)
  - **Email me your topic**
  - Create presentation (<10 min, semi-strict) + brief summary (<200 words)
    - Must present to class (email me if you're in a different timezone)
    - You can pre-record (e.g. youtube)
    - Starting next week!
  - If very few volunteers, order is random- I'll email you around at least 6 days before you present (Going to send out an email today)

Feedback / Office Hours / Other

- Tameez Latib
  - tameezlatib@gmail.com, please add “CS35L” to the subject line
  - Office Hours: Monday **4pm-6pm** (or by appointment)
  - Feedback: https://forms.gle/6kcJ2aJtzAzFMhHQ7 (anonymous google form)
- If you guys are stressed out:
  - CAPS (https://www.counseling.ucla.edu/)
    - Free with UC ship
- Something cool, store a game (snake executable) on a QR code
  - https://www.youtube.com/watch?v=ExwqNreocpg
  - (Somewhat relevant, compiling/linking, CR+LF, etc. Also example of week 10 presentation)
- Week 7...

Linking

- ❖ Recall: code is compiled in 4 steps
- ❖ Link step / linker responsible for modular code
- ❖ Takes multiple object code files and creates executable
- ❖ "Symbol resolution + relocation"
  - ➢ Symbol = variable / function
  - ➢ Resolution = reference to definition
  - ➢ Relocation = move to diff memory address

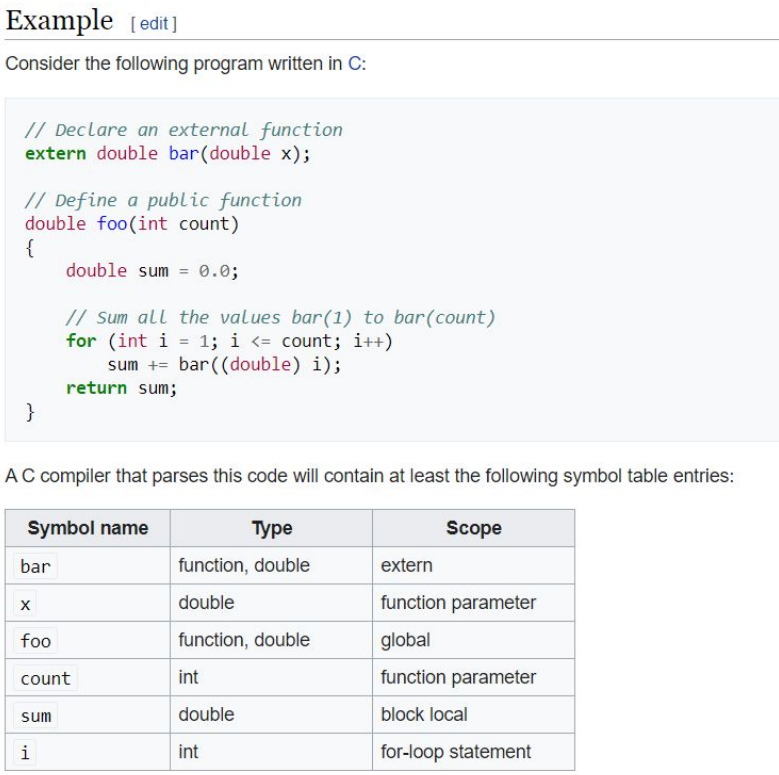


Linking

- Why?
  - Modularity (use code from other modules/people/etc)
  - Time efficiency (don’t re-compile everything)
  - Space efficiency (don’t use everything)
- How?
  - “Symbol resolution + relocation”
  - First, each variable/function has an associated ‘symbol’, associate symbols with meaning (e.g. x = 5, symbol is x, meaning is 5)
  - Relocate memory addresses so that everything fits in one executable
    - In object code, start at memory address 0, so if we have two files both at memory 0, there is a problem

Symbol resolution

- For each symbol, we have an entry in symbol table (generated by assembler)
- Example from wikipedia
- Include name + location (and maybe other stuff)
- For linking, only need global symbols
  - Global variables + functions
- If I call “bar”:
  - Linker checks symbol table,
  - finds bar, checks type, returns mem location



Example [edit]

Consider the following program written in C:

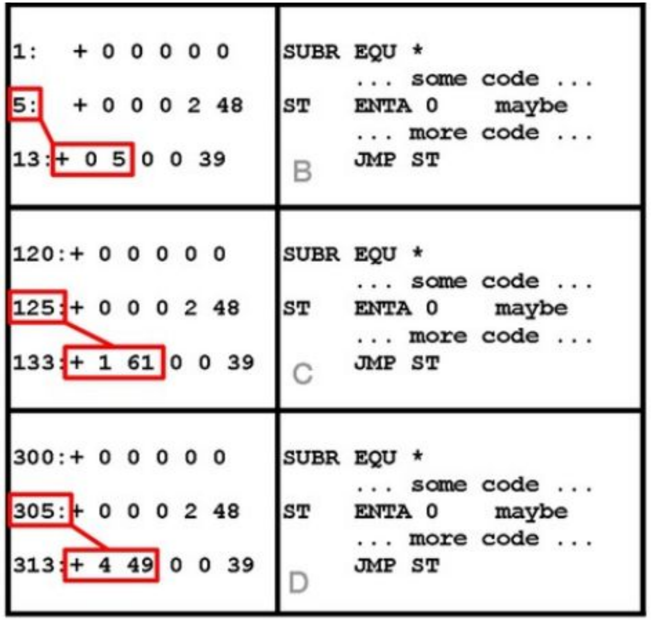
```c
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1; i <= count; i++)
        sum += bar((double) i);
    return sum;
}
```

A C compiler that parses this code will contain at least the following symbol table entries:

Symbol name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

Relocation

- Assign each symbol to unique memory address
  - Object file1 has “bar” at relative position 1, and object file2 has “foo” at relative position 1
  - Relocation step makes sure that position x refers to a unique function/variable
  - Can later be relocated at run-time too
- Example, again from wikipedia
  - Left column is machine code, right code is assembly
  - B is original object code, note starting address 1
  - C is linked object code, note starting address 120
  - D is program loaded at runtime, note starting address 300
- Here, “1 61” = 125, and “4 49” = 305



```
1:   + 0 0 0 0 0        SUBR EQU *
                             ... some code ...
5:   + 0 0 0 2 48       ST   ENTA 0     maybe
                             ... more code ...
13:+ 0 5 0 0 39              JMP ST
```

```
120:+ 0 0 0 0 0         SUBR EQU *
                             ... some code ...
125:+ 0 0 0 2 48        ST   ENTA 0     maybe
                             ... more code ...
133:+ 1 61 0 0 39            JMP ST
```

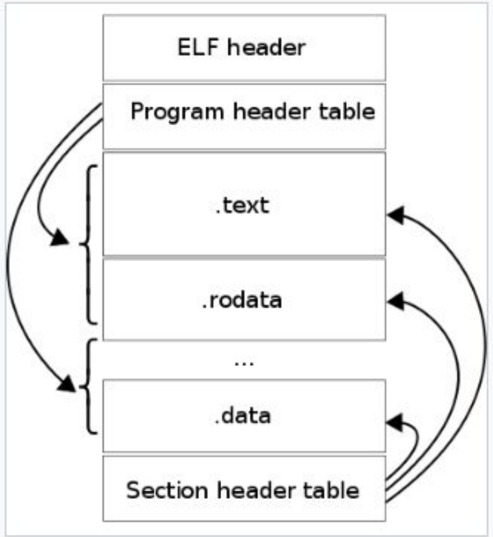
```
300:+ 0 0 0 0 0         SUBR EQU *
                             ... some code ...
305:+ 0 0 0 2 48        ST   ENTA 0     maybe
                             ... more code ...
313:+ 4 49 0 0 39            JMP ST
```

Object file

- Three types;
  - Relocatable (e.g. .o)
    - Binary code, that can be combined with other relocatable files to create an executable
    - Generated by compilers/assemblers
  - Executable (e.g. .out)
    - Binary code that can be copied into memory and executed
    - Generated by linkers
  - Shared object (e.g. .so)
    - Special version of relocatable object file that can be loaded into memory and linked dynamically at load time or run time.
    - Generated by compilers/assemblers

ELF (Executable and Linkable Format)

- Format of .o (relocatable object files)
- Try: readelf --symbols file.o vs readelf --symbols a.out
- Try: readelf -l a.out
- Section: info necessary for linking
- Segment: info necessary for runtime



An ELF file has two views: the program header shows the *segments* used at run time, whereas the section header lists the set of *sections* of the binary.

Static Linking

- Static library (e.g. .a) = collection of object code (e.g. .o)
- Roughly copy paste what is necessary from static library into your code
- Linker takes care of this, as last step (Only once!)
- To use:
  - gcc –static main.c –**L***/path/to/libraryDir* –**l***library*
  - Big L with library directory, small l with library name
  - gcc -static main.c -L. -lmyLibrary

Static Linking

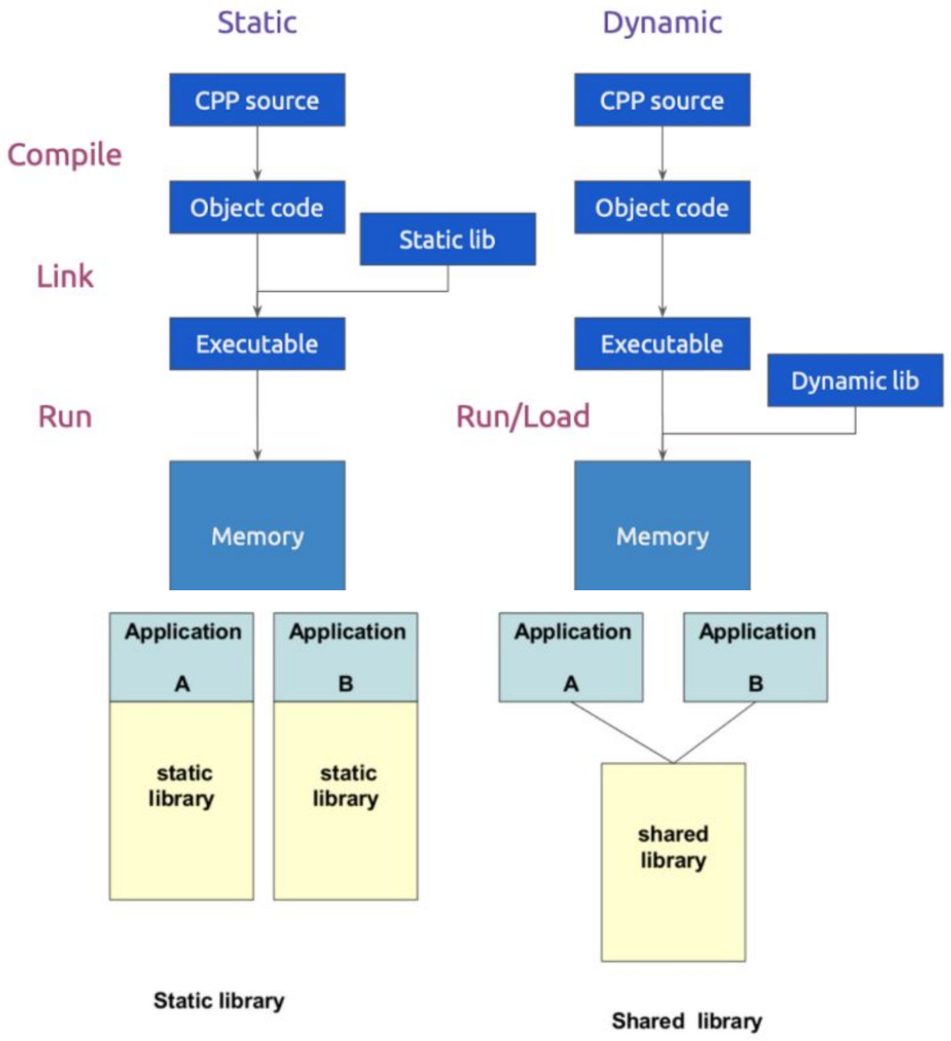
- Create lib_foo.c and lib_foo.h (source code)
- gcc -c lib_foo.c -o lib_foo.o (create object code)
- ar rcs lib_foo.a lib_foo.o (create static library)
- gcc -c main.c -o main.o (create main object code)
- gcc -static -o main main.o -L. -l_foo (create main executable using static lib)
- Note -l_foo, remove “lib” and “.a” from lib_foo.a

Dynamic Linking

- Shared library (.so) file
- At compile time, copy library references (not actual code)
- At run time, load library + run code (so here linking is at run time)
- Use gcc with the -fPIC and -shared flags
  - fPIC = position independent code
- export LD_LIBRARY_PATH=path/to/lib:$LD_LIBRARY_PATH
- Example: https://medium.com/@Cu7ious/how-to-use-dynamic-libraries-in-c-46a0f9b98270

Static vs dynamic linking

- Static
  - bigger files,
  - less runtime issues,
  - recompile if lib changes
- Dynamic
  - smaller files,
  - no recompile if lib changes,
  - slower runtime



Questions??