

CIS 330

C++ and Unix

Lecture 5

Compilers

Question 1

Which of the following methods is the correct way to declare a function pointer `funcptr` and assign the function **`double compute_average(double *y, int cnt)`** to it

1. `double (*funcptr) (double* y, int cnt); *funcptr = compute_average;`
2. `double (funcptr) (y, cnt); funcptr = compute_average;`
3. `double (funcptr) (double* y, int cnt); funcptr = &compute_average;`
4. `double (*funcptr) (double* x, int cnt); funcptr = &compute_average;`

Question 2

Which of the following is **NOT** correct?

1. In some cases, we can use a function that takes in a 2-D matrix as input (e.g., `int** some_array`) for processing a 1-D matrix (e.g., `int* some_other_array`)
2. We can store a 2-D array as a 1-D array
3. A 2-D matrix on the heap stores all of its data consecutively in memory (just like a 1-D matrix)
4. We can pass a function pointer as a parameter to another function

Question 3

What will be the output when the following code is executed?

```
int some_func(int* i) {  
  
    int j = *i;  
  
    return j++;  
  
}  
  
int main() {  
  
    int i = 101;  
  
    some_func(&i);  
  
    fprintf(stdout, "%d\n", i);  
  
}
```

1. 101
2. some address
3. 102
4. undefined behavior

Question 4

Given the following piece of code, which of the following statements is correct?

```
void my_malloc(double** b) {  
  
    double* b_tmp;  
  
    b_tmp = (double*) malloc(sizeof(double) * 128);  
  
    assert(b_tmp);  
  
    fprintf(stdout, "My address is %p\n", b_tmp);  
  
    *b = b_tmp;  
  
}
```

1. b and b_tmp have different data types, so it will not return a valid address to the main function (where my_malloc() was called)
2. It behaves like a smart malloc() function, where the return value from malloc() is checked to see if it's valid
3. None of the above are true.
4. Since malloc'd memory was assigned to a variable created inside of a function (i.e., b_tmp is on the stack), it will be lost when the function returns

Questions?

Homework 2

allocating memory

```
int main() {
```

```
    int* some_array;
```

```
    allocate_mem(...);
```

```
}
```

```
void allocate_mem(...) {
```

```
}
```

Homework 2

allocating memory

```
int main() {
```

```
    int* some_array;
```

```
    allocate_mem(&some_array); // pass by reference
```

```
}
```

```
void allocate_mem(...) {
```

```
}
```


Homework 2

allocating memory

```
int main() {  
    int* some_array;  
    allocate_mem(&some_array); // pass by reference  
}  
  
void allocate_mem(int** array) {  
  
}
```

Homework 2

allocating memory

```
int main() {  
    int* some_array;  
    allocate_mem(&some_array); // pass by reference  
}
```

```
void allocate_mem(int** array) {  
    *array = (int*) malloc(sizeof(int) * 100);  
}
```

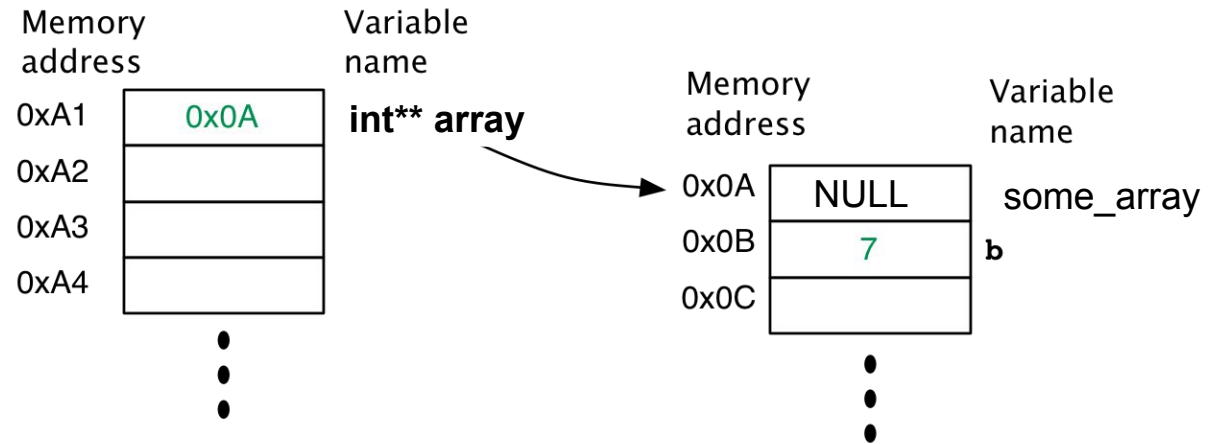
Homework 2

allocating memory

```
int main() {  
    int* some_array;  
    allocate_mem(&some_array); // pass by reference  
}
```

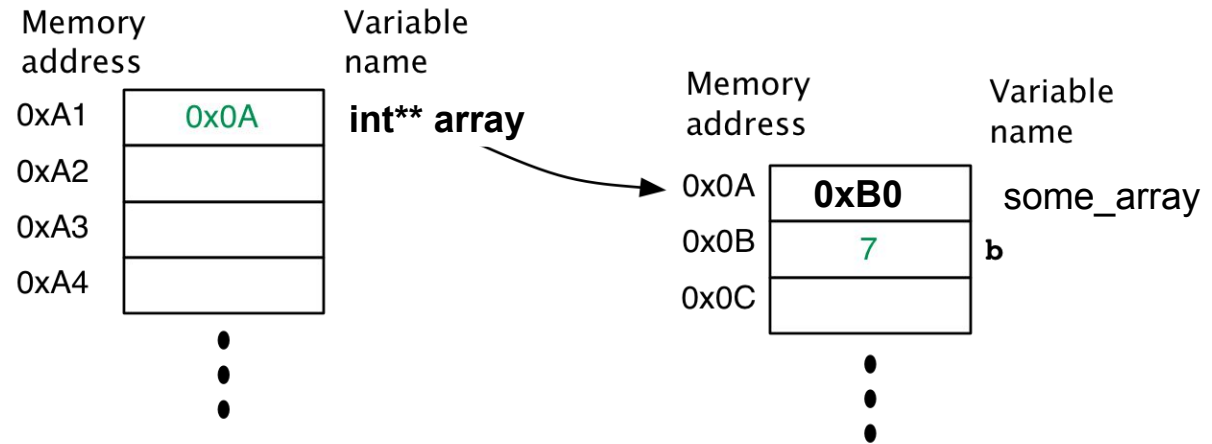
```
void allocate_mem(int** array) {  
    int* tmp = (int*) malloc(sizeof(int) * 100);  
    *array = tmp;  
}
```

Pointers



```
int main() {  
    int* some_array = NULL;  
    allocate_mem(&some_array); // pass by reference  
}  
  
void allocate_mem(int** array) {  
    *array = (int*) malloc(sizeof(int) * 100);  
}
```

Pointers



```
int main() {  
    int* some_array = NULL;  
    allocate_mem(&some_array); // pass by reference  
}  
  
void allocate_mem(int** array) {  
    *array = (int*) malloc(sizeof(int) * 100);  
}
```

Questions?

Modular Programming

- C is a functional language – modularize your code into functions!
- Reduces the amount of code you need to write, and makes debugging easier

Code Spanning Multiple Files

- You could write your entire program in a single, very large, .c file
- However, it's better to separate your code into multiple files
 - Easier to keep track of your code
 - Easier to compile (more on this later)
 - Easier to collaborate (e.g., via Git)

Code Spanning Multiple Files

- First file – contains the main function
- Second file(s)
 - .c file that contains the code
 - .h file that contains the function declaration (header file)
 - `stdio.h`, `stdlib.h`, and `string.h` are examples of header files
 - Interface to using the functions written in the .c file
- Third file(s), etc.
- Header files should be **included carefully** to avoid multiple inclusions
 - use if-not-defined check if a header file is already included

Header File Example

arithmetic.h

1. `int add_two_numbers(int a, int b);`

numbers.c

2. `#include "arithmetic.h"`

3. `int main()`

4. `{`

5. `int a = 1;`

6. `int b = 3;`

7. `int c = add_two_numbers(a, b);`

8. `fprintf(stdout, "%d + %d = %d\n", a, b, c);`

9. `return 0;`

10. `}`

Examples of if-not-defined

1. `#ifndef ARITHMETIC_H`
 2. `#define ARITHMETIC_H`
 3. `int add_two_numbers(int a, int b);`
 4. `#endif /* not defined ARITHMETIC_H */`
- The first time arithmetic.h is called, it defines the variable ARITHMETIC_H and the regular definitions contained within arithmetic.h
 - Next time arithmetic.h is called (redundantly), ARITHMETIC_H will already be defined, and the content within the if-not-defined will be skipped

Compiling Multiple Files

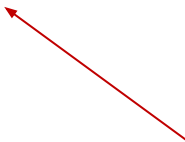
- The simple method – just list all the files
 - e.g., `> gcc arithmetic.c numbers.c`
 - With this method, we have to compile both files every time we make changes to one or the other – not much better than having one large file
- Compiling the code separately
 - Compile each .c file into **object files**, and then **link** them together

Linker

```
> gcc -c arithmetic.c  
> gcc -c numbers.c  
> gcc arithmetic.o numbers.o
```



Compile the code into object files



Link the object files together to create the executable

- If we change one file (e.g., arithmetic.c), we only have to recompile arithmetic.c then re-link the object files

Compiler Options

- -o -> specify the executable name
 - e.g., `gcc -o run arithmetic.o numbers.o`
- -Wall -> enable all compiler warning messages.
 - It is -W with all option
 - You can use -W to enable/disable specific warnings
- -O# -> set the compiler optimization level
 - e.g., -O3
- -std -> sets the C standard to follow
 - e.g., -std=c11 (follow the C11 standard)
- -g -> enable debugging (more on this later)

Optimization Level

Option	Optimization
-O0	Optimize for compile time (no optimization, default)
-O1 or -O	Moderate optimization – optimizes reasonably well but does not degrade compilation time
-O2	Full optimization – highly optimized code and slowest compilation time
-O3	-O2 + aggressive subprogram inlining and vectorization
-Os	Optimize for code size
-Ofast	-O3 + non-accurate math calculation (floating point roundoff error)

Compiler Driver

- gcc is actually a compiler driver – it invokes several “tools” to accomplish the task of converting source code to executable code

- For example,

> gcc arithmetic.c numbers.c

Invokes

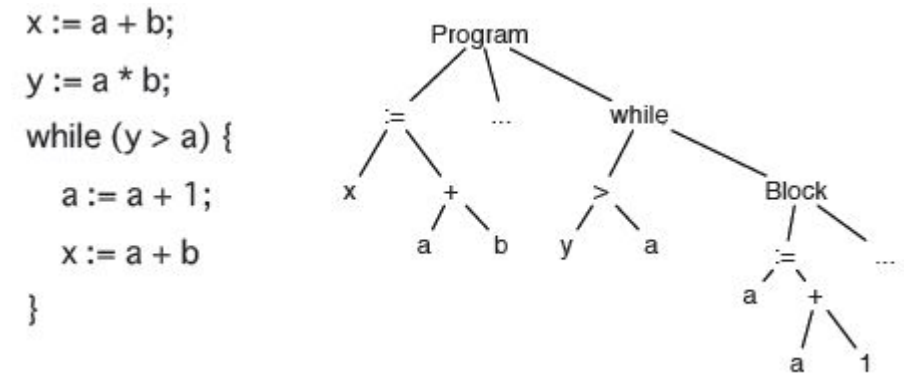
1. Preprocessor
2. Compiler (cc1)
3. Assembler (as)
4. Linker (ld)

Preprocessor

- Macro processor that transforms your code:
 - Includes header files
 - Macro expansion
 - Removes comments
 - And more
- Output typically looks similar to the input (i.e., source code)
 - Input - C language
 - Output - C language (just a bit longer)

Compiler

- Compiler takes preprocessed C language file and generates assembly code
 - cc1 contains the preprocessor and the compiler
- Compilation stages consist of
 - Front end
 - Middle end
 - Back end



- Front end
 - Parses the source code to generate abstract syntax tree (AST)
 - Data structure that is the tree representation of the abstract syntactic structure of the source code
- Middle end
 - Converts AST to different representations for optimization
 - Generates register-transfer language (RTL)
 - RTL is a hardware-specific representation that corresponds to some abstract target architecture (e.g., with infinite number of registers)
- Back end
 - Generates assembly code for the target architecture

Output - English-readable assembly language

Assembler

- Converts assembly language to object code
- Object code is in binary (but readable with tools such as *objdump*)

Linker

- “Merges” object files into a single executable object file
- As part of the merging process, it resolves external references
 - e.g., you can compile your code using *fprintf* without knowing exactly how *fprintf* is implemented
 - However, when you want to actually run this code, you must know where this piece of code is located (i.e., in an external library).
- “Relocates” symbols from their relative position in the object files to absolute position in the executable, and updates their references (i.e., use) to this new position
 - It “copy & paste” the *fprintf* function from the original objective file to the executable

Executable and Linkable Format (ELF)

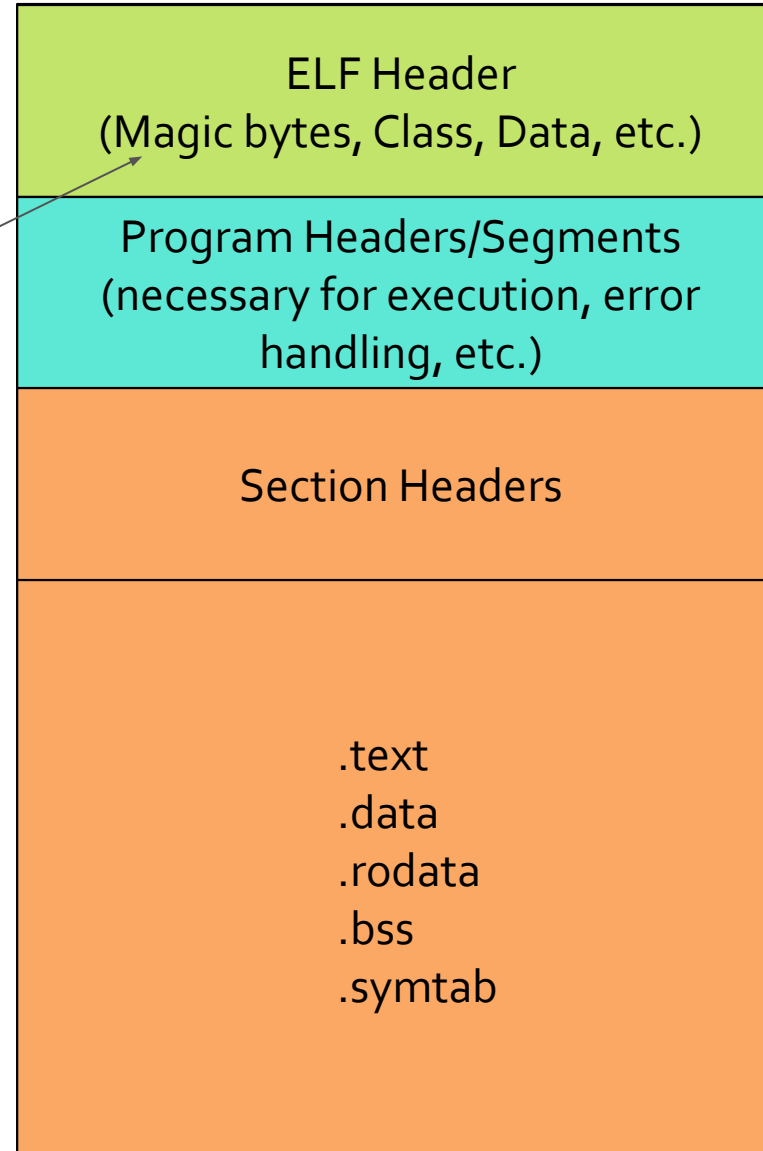
- Common standard binary file format for executables, object code, etc.
- Derives from the AT&T System V Unix OS

ELF Layout

Class - tells us information about the target architecture (e.g., 32-bit or 64-bit?)

Data - whether the data is stored in LSB or MSB
LSB - little Endian
MSB - big Endian

0x7fELF



ELF Layout

Program headers describe how to create a process/memory image for runtime execution

Section headers defines all the sections contained within the ELF file.

The headers are used for linking and relocation.

ELF Header
(Magic bytes, Class, Data, etc.)

Program Headers/Segments
(necessary for execution, error handling, etc.)

Section Headers

.text
.data
.rodata
.bss
.symtab

ELF Layout

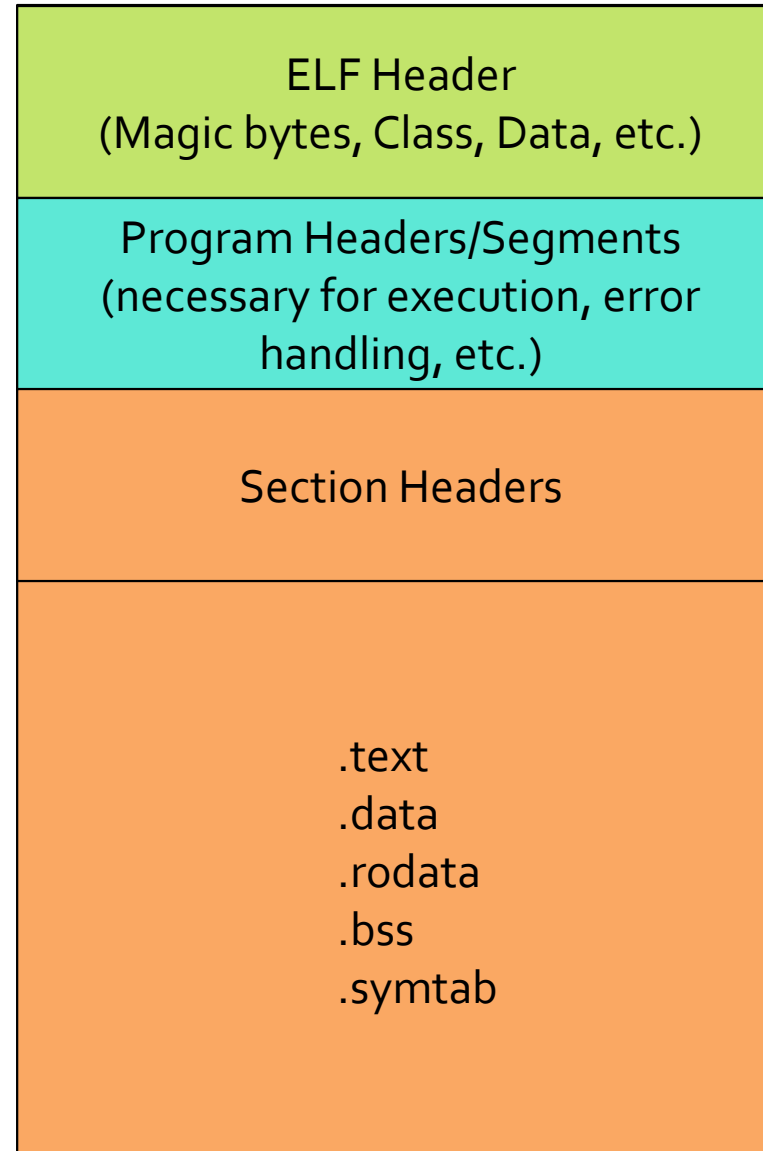
.text contains the executable code, packed into a segment with read and execute access rights. It is loaded only once since the content shouldn't change.

.data contains initialized data (i.e., static), with read and write access.

.rodata also has initialized data, but with only read access

.bss contains uninitialized data, and therefore has both read and write access.

.symtab contains the global symbol (i.e., functions and variables) table



Example

main.c

```
1.  int e = 7;  
2.  int main()  
3.  {  
4.      int r = a();  
5.      return 0;  
6.  }
```

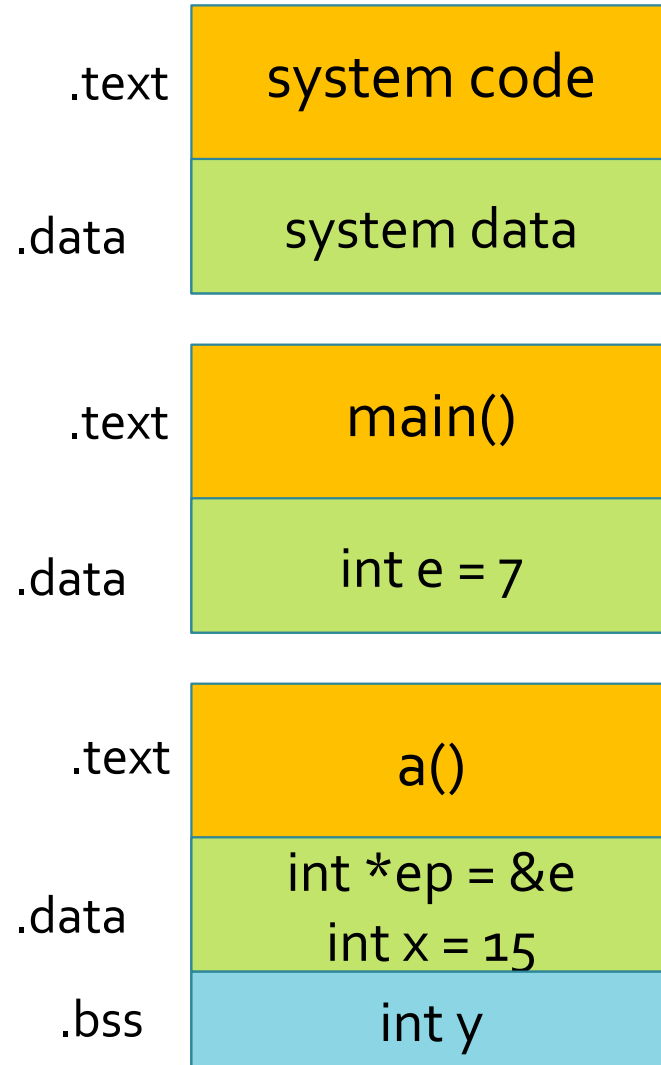
a.c

```
1.  extern int e;  
2.  int *ep = &e;  
3.  int x = 15;  
4.  int y;  
5.  int a()  
6.  {  
7.      return *ep + x + y;  
8.  }
```

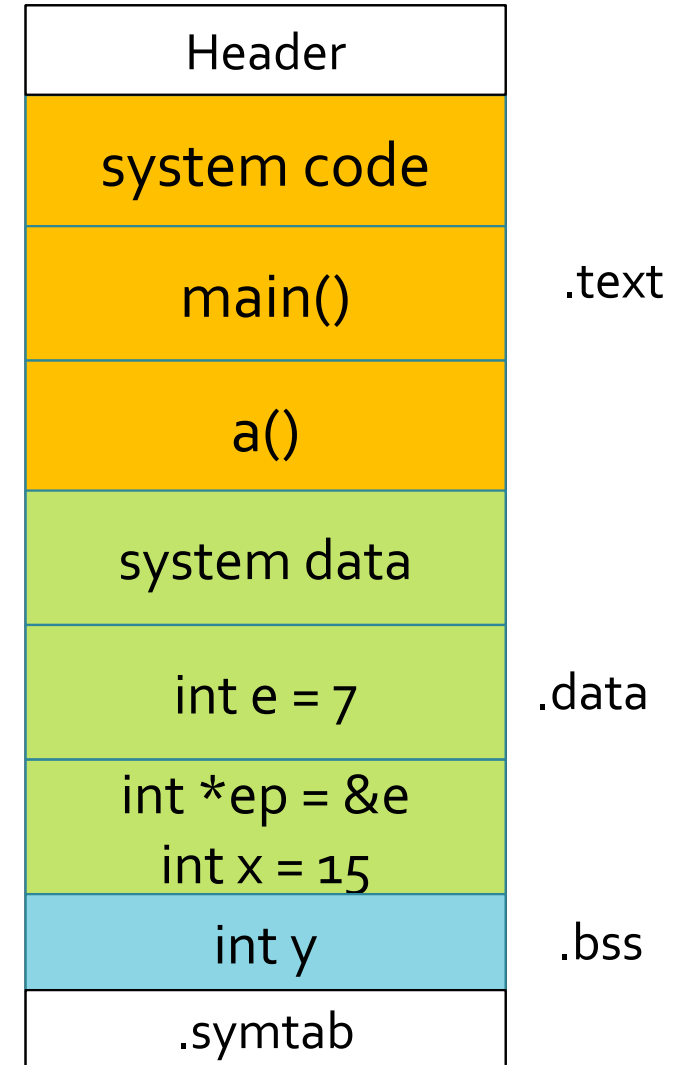
Each file compiles to its respective object files

Example

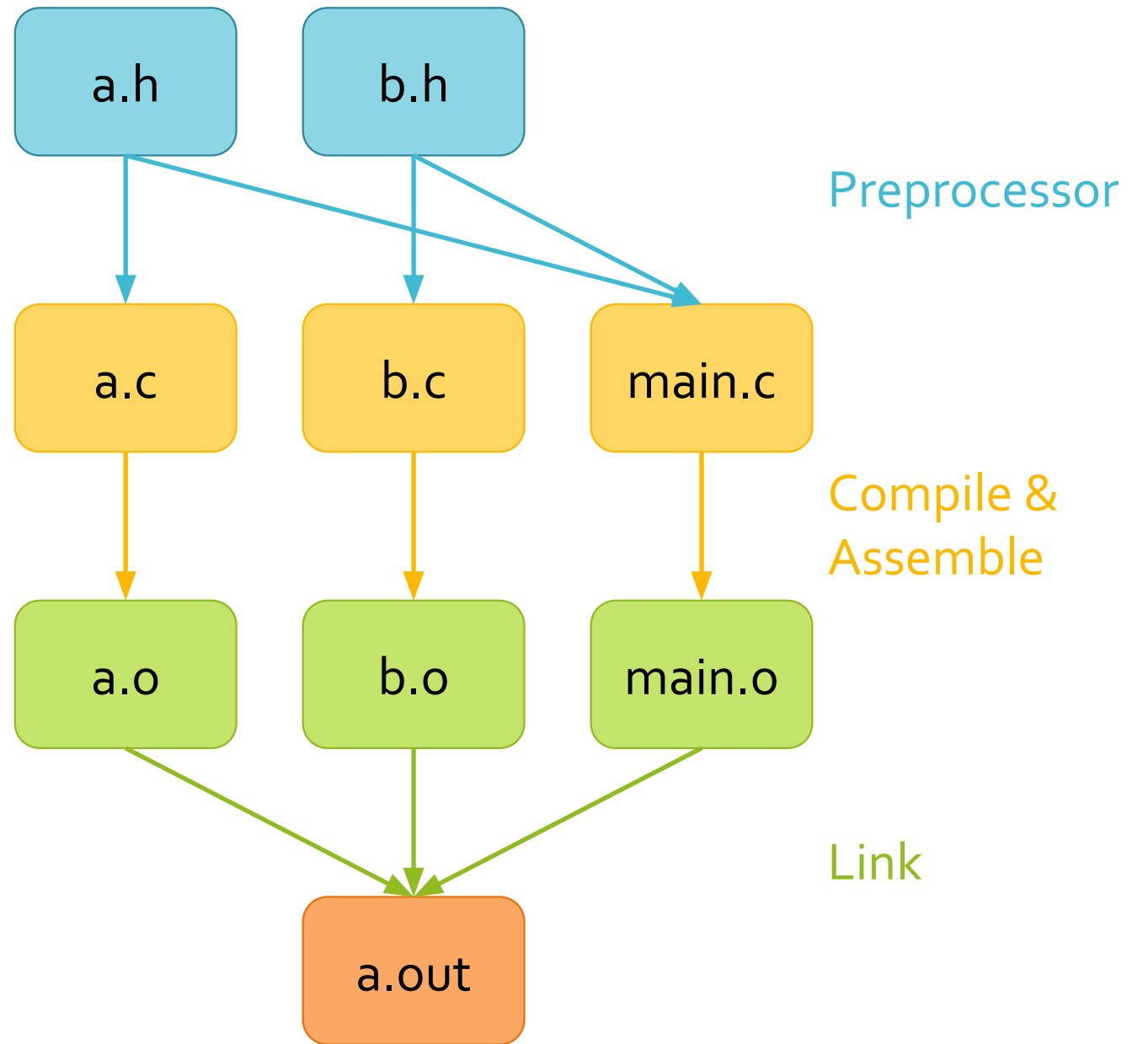
Relocatable Object Files



Executable Object Files



Overview



Questions?

Symbols

- Symbols can be either **strong** or **weak**
 - Strong – **functions** and **initialized global variables**
 - Weak – **uninitialized global variables**
- Strong symbols can only appear once
- A weak symbol can be “overridden” by a strong symbol (of the same name) – references to the weak symbols are actually referencing the strong symbol
- Multiple weak symbols can be chosen arbitrarily by the linker
- Be careful with global variables!

Example

File 1

```
int x;  
p1() {}
```

```
int x;  
p1() {}
```

```
int x;  
int y;  
p1() {}
```

```
int x = 5;  
int y = 7;  
p1() {}
```

```
int x = 5;  
p1() {}
```

File 2

```
p1() {}
```

```
int x;  
p2() {}
```

```
double x;  
p2() {}
```

```
double x;  
p2 () {}
```

```
int x;  
p2 () {}
```

Example

File 1

```
int x;  
p1() {}
```

```
int x;  
p1() {}
```

```
int x;  
int y;  
p1() {}
```

```
int x = 5;  
int y = 7;  
p1() {}
```

```
int x = 5;  
p1() {}
```

File 2

```
p1() {}
```

```
int x;  
p2() {}
```

```
double x;  
p2() {}
```

```
double x;  
p2 () {}
```

```
int x;  
p2 () {}
```

Link time error – two strong symbols

Example

File 1

```
int x;  
p1() {}
```

```
int x;  
p1() {}
```

```
int x;  
int y;  
p1() {}
```

```
int x = 5;  
int y = 7;  
p1() {}
```

```
int x = 5;  
p1() {}
```

File 2

```
p1() {}
```

```
int x;  
p2() {}
```

```
double x;  
p2() {}
```

```
double x;  
p2 () {}
```

```
int x;  
p2 () {}
```

Link time error – two strong symbols

References to x will be to the same variable

Example

File 1

```
int x;  
p1() {}
```

```
int x;  
p1() {}
```

```
int x;  
int y;  
p1() {}
```

```
int x = 5;  
int y = 7;  
p1() {}
```

```
int x = 5;  
p1() {}
```

File 2

```
p1() {}
```

```
int x;  
p2() {}
```

```
double x;  
p2() {}
```

```
double x;  
p2 () {}
```

```
int x;  
p2 () {}
```

Link time error – two strong symbols

References to x will be to the same variable

Write to x in file2 **may** overwrite y

Example

File 1

```
int x;  
p1() {}
```

```
int x;  
p1() {}
```

```
int x;  
int y;  
p1() {}
```

```
int x = 5;  
int y = 7;  
p1() {}
```

```
int x = 5;  
p1() {}
```

File 2

```
p1() {}
```

```
int x;  
p2() {}
```

```
double x;  
p2() {}
```

```
double x;  
p2 () {}
```

```
int x;  
p2 () {}
```

Link time error – two strong symbols

References to x will be to the same variable

Write to x in file2 **may** overwrite y

Write to x in file2 **will** overwrite y

Example

File 1

```
int x;  
p1() {}
```

```
int x;  
p1() {}
```

```
int x;  
int y;  
p1() {}
```

```
int x = 5;  
int y = 7;  
p1() {}
```

```
int x = 5;  
p1() {}
```

File 2

```
p1() {}
```

```
int x;  
p2() {}
```

```
double x;  
p2() {}
```

```
double x;  
p2 () {}
```

```
int x;  
p2 () {}
```

Link time error – two strong symbols

References to x will be to the same variable

Write to x in file2 **may** overwrite y

Write to x in file2 **will** overwrite y

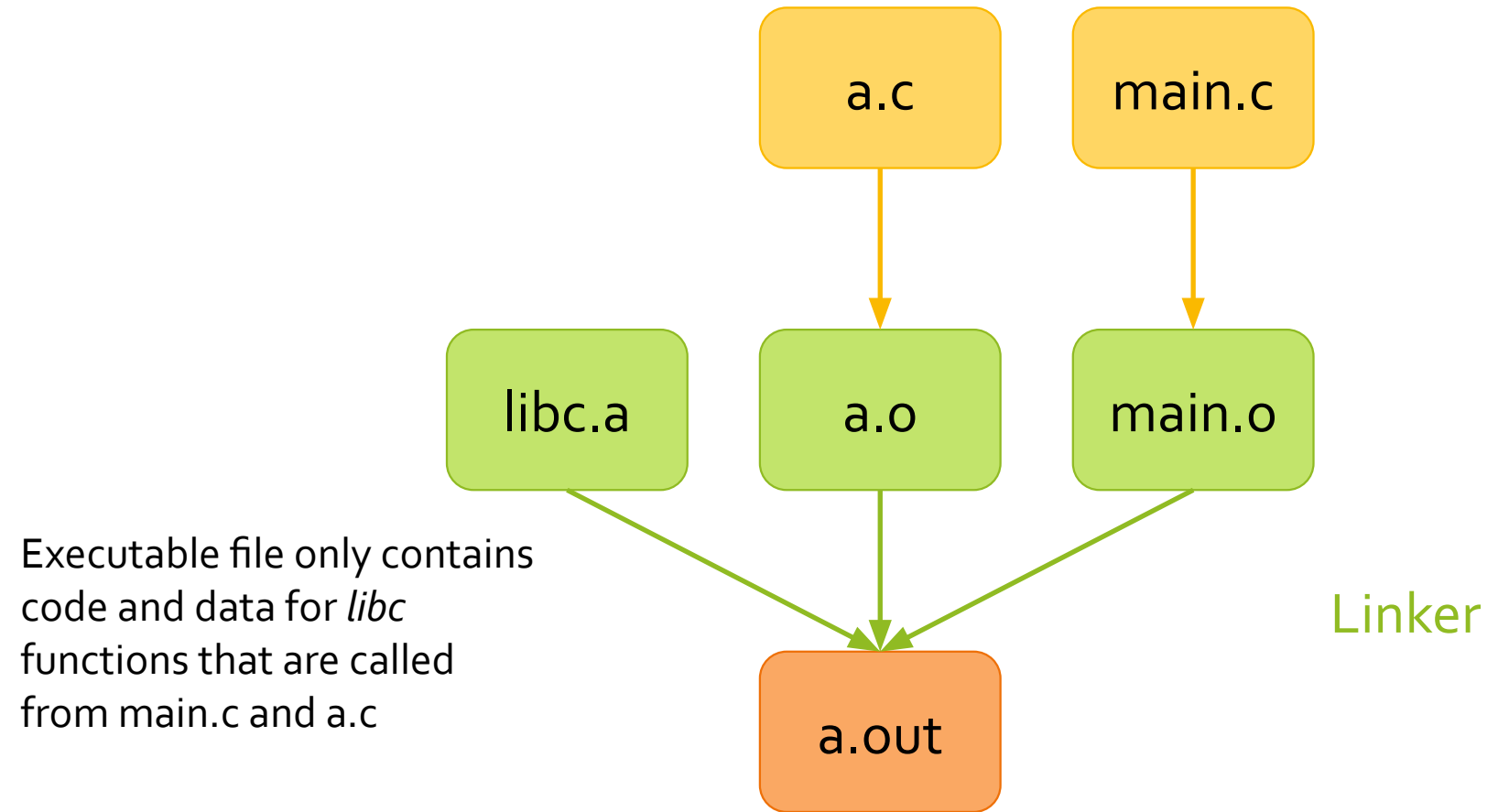
References to x will be to the same initialized variable

Questions?

Libraries

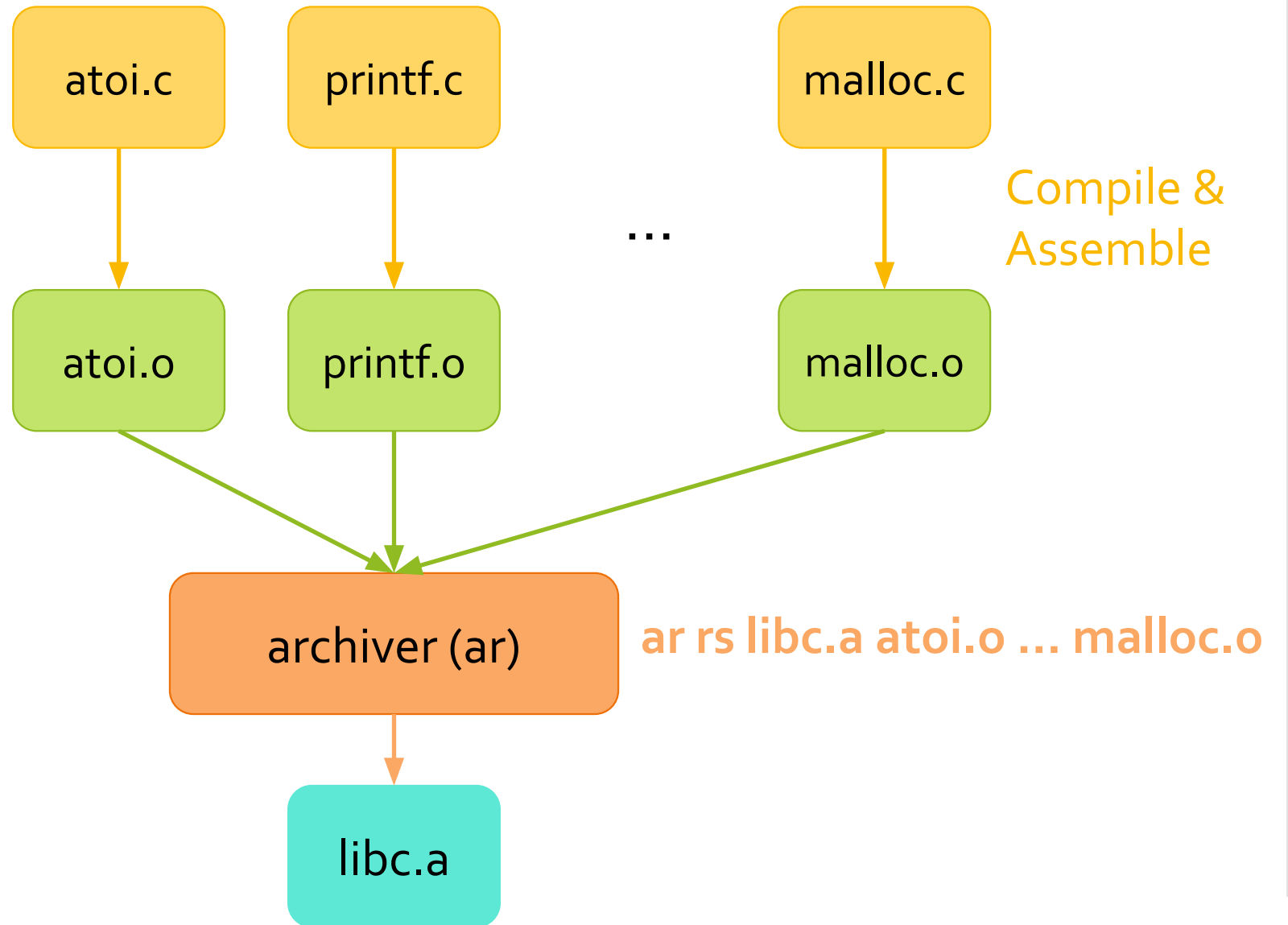
- How should we package commonly used functions?
 - e.g., sorting, string parsing, etc.
- Put these functions in a single source file -> link
 - Object files are big -> space and time consuming
- Put each function in a different file -> link
 - Too much work to link every needed function/file
- Static Libraries (.a archive files)
 - **Concatenate** related **relocatable object files** into a single file with an index (called an archive)
 - Helps the linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives
 - If an archive member file resolves reference, link it into the executable

Static Libraries



- Linker selectively concatenates only the .o files in the archive that are actually needed by the program

Creating Static Libraries



Archiver

- Creates static libraries by combining individual files/functions
- Allows **incremental** updates – recompile functions that changes and replace .o file in the archive
- Commonly used libraries
 - C standard library (libc.a) – 8 MB of 900 object files
 - I/O, memory allocation, string handling, etc.
 - C math library (libm.a) – 1 MB archive of 226 object files
 - floating point math (e.g., sin, cos, tan, exp, sqrt, etc.)

Using Libraries

- Use the option `-L<path to file containing library> -l<library name>`
- `gcc -o myapp main.c -L/home/jeec/lib -lmylib`
- Libraries sometimes need header files (so that you know **how** the functions are used)
- `-I<path to header file>`
- `LD_LIBRARY_PATH` can be used to specify the library directories (so that you don't have to explicitly use the `-L` option)
 - `export LD_LIBRARY_PATH=/usr/lib`
 - `export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH` and so on...

Questions?

Library Ordering

- Ordering of the object files and library matters!
- `gcc main.c -L/home/lib -lex1 -lex2 -lex3`
- Linker maintains a symbol table:
 - a list of symbols exported (provided) by all object and library files it has seen
 - a list of symbols requested (by your code) but not found (**undefined**)

Library Ordering

- When the linker encounters a new object file
 - a new list of symbols (i.e., functions that this file implements) is exported
 - if it's one of the symbols in the **undefined** list, remove it from the undefined list (no longer **undefined**)
 - If an identical (strong) symbol has already been encountered, throw an error (different objects with the **same symbol**)
 - a new list of imported symbols (maybe one of the functions requires yet another function) goes to the **undefined** list (unless already on the existing list of symbols)
 - Object files are concatenated into the executable - any previously exported symbol can be found later
- When the linker encounters a new **library**
 - scan each object file in the library
 - if **any** of the symbols exported by the library is in the undefined list, add the object, and it is processed like a regular object file
 - if **any** of the objects in the library has been added, the library is scanned **again** (an object added later might be using a call in an object scanned earlier)
- At the end, it looks at the **undefined** list – if something is on it, thrown an error (undefined reference).
- Note – **Linker does not go back to a library once it has finished scanning it**

Library Ordering

- Why would the ordering matter?
- Circular dependency?
 - A needs something from B, and B needs something from A
 - What happens?

Library Ordering Example

add.c

```
1. #include "sub.h"
2. int add_numbers(int a, int b)
3. {
4.     int c = sub_numbers(a, -1 * b);
5.     return c;
6. }
7.
```

subtract.c

```
1. #include "add.h"
2. int sub_numbers(int a, int b)
3. {
4.     int c = add_numbers(a, -1 * b);
5.     return c;
6. }
```

main.c

```
#include "add.h"

int main(int argc, char** argv)
{
    add_numbers(1, 2);
    return 0;
}
```

Library Ordering Example

add.c

```
1. #include "sub.h"
2. int add_numbers(int a, int b)
3. {
4.     int c = sub_numbers(a, -1 * b);
5.     return c;
6. }
7.
```

subtract.c

```
1. #include "add.h"
2. int sub_numbers(int a, int b)
3. {
4.     int c = add_numbers(a, -1 * b);
5.     return c;
6. }
```

main.c

```
#include "add.h"

int main(int argc, char** argv)
{
    add_numbers(1, 2);
    return 0;
}
```

- Which will work?
 - gcc -o run main.o -L. -lsub -ladd
 - gcc -o run main.o -L. -ladd -lsub
- Why?

Library Ordering Example

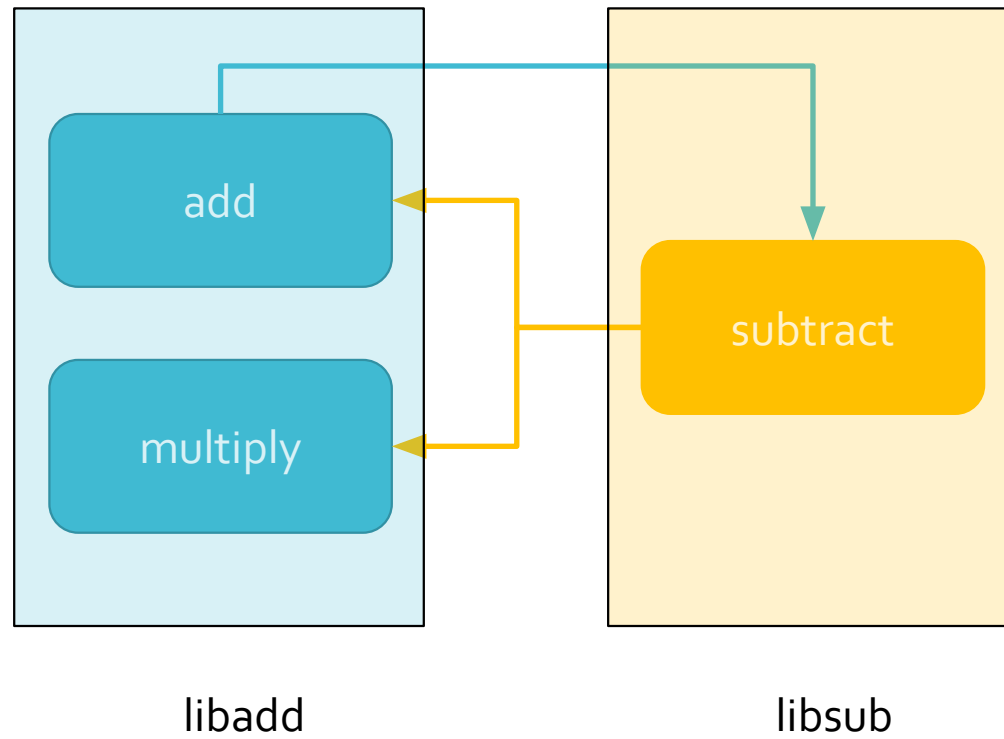
- Which will work?
 - `gcc -o run lectureo6.o -L. -lsub -ladd`
 - `gcc -o run lectureo6.o -L. -ladd -lsub`
- Why?
 - Remember the rules above for linking
 - First case
 - Reads main
 - -> add_numbers in undefined list
 - Reads subtract library
 - -> exports subtract_numbers
 - -> not in undefined list
 - -> skips
 - Reads add library
 - -> exports add_numbers
 - -> in undefined list (main requires it)
 - -> adds add_number to symbol list and remove is it from undefined
 - -> imports subtract_numbers and adds it to undefined list
 - Undefined reference to subtract_numbers remains (linker error)

Library Ordering Example

- Which will work?
 - `gcc -o run lecture06.o -L. -lsub -ladd`
 - `gcc -o run lecture06.o -L. -ladd -lsub`
- Why?
 - Remember the rules above for linking
 - Second case
 - Reads main
 - -> add_numbers in undefined list
 - Reads add
 - -> exports add_numbers
 - -> in undefined list
 - -> adds add_number to symbol list and remove is it from undefined
 - -> imports subtract_numbers and adds it to undefined list
 - Reads subtract
 - -> exports subtract_numbers
 - -> in undefined list
 - -> adds subtract_numbers to symbol list and removes it from undefined
 - Nothing in undefined list (done)

Library Ordering Example 2

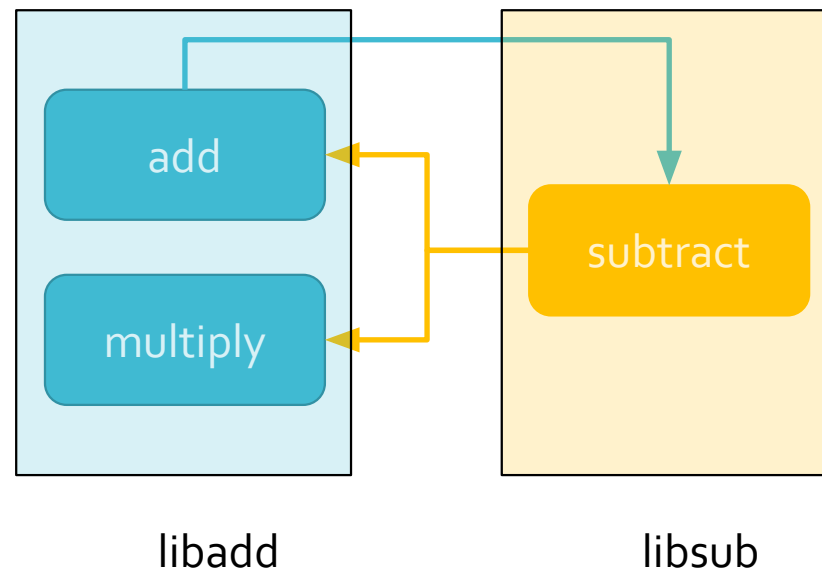
- What happens if...



Library Ordering Example 2

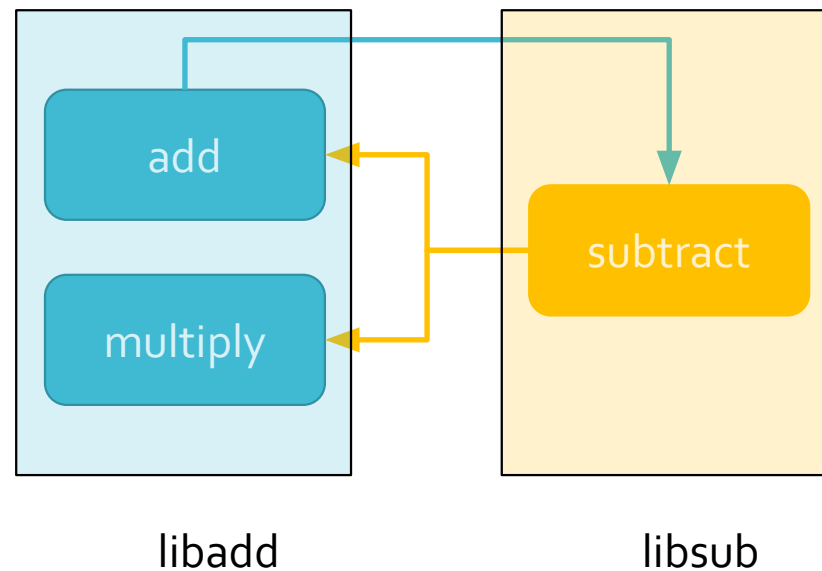
- Case 1 - gcc -o run lectureo6.o -L. -lsub -ladd
- Case 2 - gcc -o run lectureo6.o -L. -ladd -lsub

Library Ordering Example 2



- Case 1 - `gcc -o run lecture06.o -L. -lsub -ladd`
 - main -> exports add
 - sub -> exports subtract -> not in the undefined list -> skip
 - add -> exports add and multiply -> add is in the undefined list -> process add.o -> exports add and removes it from undefined list -> imports subtract and adds it to undefined list -> scan again -> nothing new is found
 - Undefined reference to `subtract_numbers`

Library Ordering Example 2



- Case 2 - `gcc -o run lectureo6.o -L. -ladd -lsub`
 - main -> exports add
 - add -> exports add and multiply -> add is in the undefined list -> process add.o -> exports add and removes it from undefined list -> imports subtract and adds it to the undefined list -> scan again (because something was added) -> nothing new is found
 - sub -> exports subtract -> subtract is in the undefined list process subtract.o -> exports subtract and removes it from undefined list -> imports multiply and add -> add is resolved and multiply is placed in the undefined list
 - Undefined reference to `multiply_numbers`

Library Ordering Example 2

- How would we fix this?

Library Ordering Example 2

- How would we fix this?
 - You can specify a library more than once

Library Ordering Example 2

- How would we fix this?
 - You can specify a library more than once
- Which would work?
 - `gcc -o run lectureo6.o -L. -ladd -lsub -ladd`
 - `gcc -o run lectureo6.o -L. -lsub -ladd -lsub`

Creating a Library

Static library

```
gcc -c add.c -o add.o  
gcc -c sub.c -o sub.o  
ar rcs libmymath.a add.o sub.o
```

Dynamic library

```
gcc -fPIC -c *.c  
gcc *.o -shared -o liball.so
```

Dynamic Libraries

- Difference between static and dynamic
 - Static – everything is include in the executable (you don't need to go searching for your library)
 - Dynamic – it exists as a separate file (you DO need to go searching for your library e.g., LD_LIBRARY_PATH)
 - Pros and cons – dynamic library reduces the file size, and allows you to just recompile the library if changes are made. On the other hand, static libraries are faster at run-time, and less susceptible to breaking.

Makefiles

- Convenient way to compile your code without having to type out everything every time
- Create a file named 'Makefile'
- Type 'make'
- <executable/target name> : pre-requisites (required files)
recipe (rules for making the target)

Makefiles

- Defines a set of commands and rules for compiling your code
- Useful commands and rules
 - `$(wildcard pattern)` – space separated list of file names that matches the *pattern*
 - `$(addsuffix suffix, names...)` – *names* is a series of file names, and *suffix* is appended to the end of each name
 - `$(basename names...)` – extracts all but the suffix of each file name in *names*
 - `%` – Pattern rule `%` refers to exactly one (i.e., exactly one file with extension `.c`
 - `%.o : %.c` – pre-requisite for a `.o` file is the corresponding `.c` file
 - `%.o : %.c $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@`
 - `%@` and `$<` are used to substitute the names of the **target** and **source** file in each case.
 - Generate `a.c` from `a.o` using `gcc -c <some flags> a.c -o a.o`