CSE 333
Section 3

Makefiles, C++ Intro, HW2 Overview

W UNIVERSITY of WASHINGTON

Checking In & Logistics

Quick check-in:

Do you have any questions, comments, or concerns?

Exercises going ok?

Lectures making sense?

REMINDERS:

Exercise 9: Due Monday (7/15) @ 10:00 am

Exercise 10: Due Wednesday (7/17) @ 10:00 am

Homework 2: Due Thursday (7/18) @ 11:00 pm

Makefile Demo

Exercise 1

Pointers, References, & Const

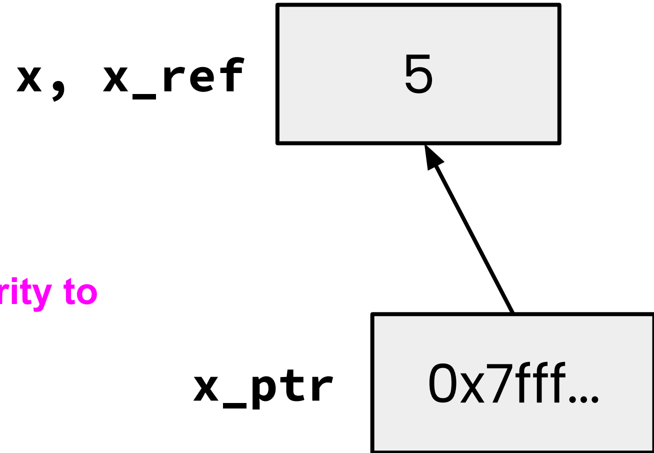
Example

Consider the following code:

```
int x = 5;
int& x_ref = x;
int* x_ptr = &x;
```

Note syntactic similarity to pointer declaration

Still the address-of operator!



What are some tradeoffs to using pointers vs references?

Pointers vs. References

Pointers

- Can move to different data via reassignment/pointer arithmetic
- Can be initialized to **NULL**
- Useful for output parameters: `MyClass* output`

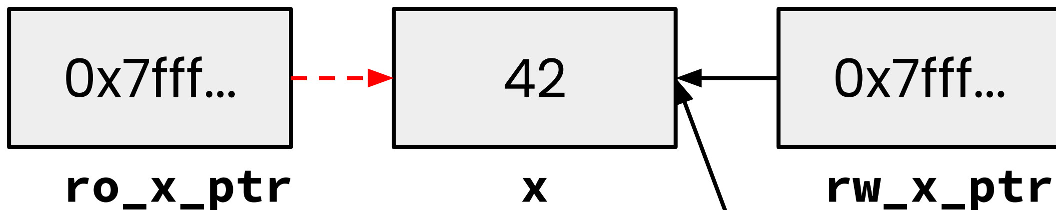
References

- References the same data for its entire lifetime - *can't reassign*
- No sensible "default reference," must be an alias
- Useful for input parameters: `const MyClass &input`

Pointers, References, Parameters

- `void func(int& arg)` vs. `void func(int* arg)`
- Use references when you don't want to deal with pointer semantics
  - Allows real pass-by-reference
  - Can make intentions clearer in some cases
- **STYLE TIP:** use references for input parameters and pointers for output parameters, with the output parameters declared last
  - Note: A reference can't be `NULL`

Const



- Mark a variable with `const` to make a compile time check that a variable is never reassigned
- Does not change the underlying write-permissions for this variable

```
int x = 42;

// Read only
const int* ro_x_ptr = &x;

// Can still modify x with
rw_x_ptr!
int* rw_x_ptr = &x;

// Only ever points to x
int* const x_ptr = &x;
```

0x7fff...

x_ptr

Legend

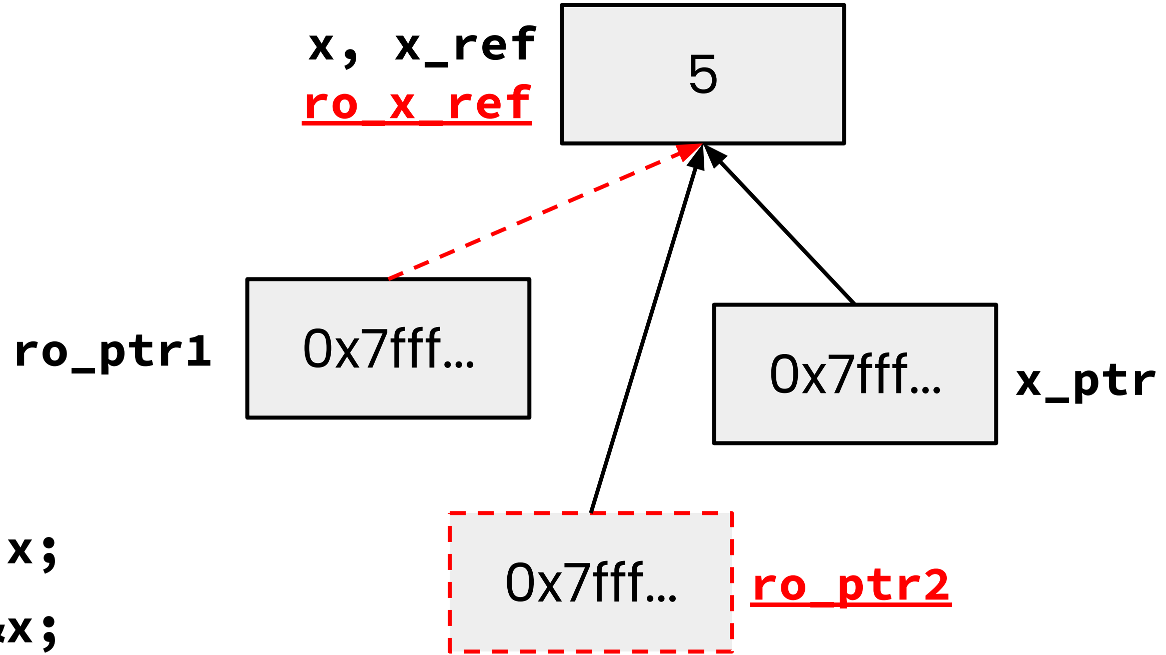
Red = can't change box it's next to

Black = read and write

Exercise 2

Exercise 6

```
int x = 5;
int& x_ref = x;
int* x_ptr = &x;
const int& ro_x_ref = x;
const int* ro_ptr1 = &x;
int* const ro_ptr2 = &x;
```



"Pointer to a const int"

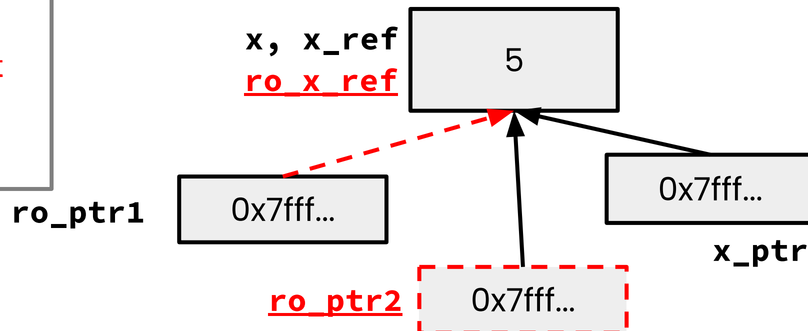
"Const pointer to an int"

Tip: Read the declaration "right-to-left"

Legend
Red = can't change box it's next to
Black = read and write

Exercise 6

Legend
Red = can't change box it's next to
Black = "read and write"




```
void foo(const int& arg);
void bar(int& arg);
```

```
int x = 5;
int& x_ref = x;
int* x_ptr = &x;
const int& ro_x_ref = x;
const int* ro_ptr1 = &x;
int* const ro_ptr2 = &x;
```

Which lines result in a compiler error?

✔ OK ❌ ERROR

- ✔ `bar(x_ref);`
- ❌ `bar(ro_x_ref);` ro_x_ref is const
- ✔ `foo(x_ref);`
- ✔ `ro_ptr1 = (int*) 0xDEADBEEF;`
- ❌ `x_ptr = &ro_x_ref;` ro_x_ref is const
- ❌ `ro_ptr2 = ro_ptr2 + 2;` ro_ptr2 is const
- ❌ `*ro_ptr1 = *ro_ptr1 + 1;` (*ro_ptr1) is const

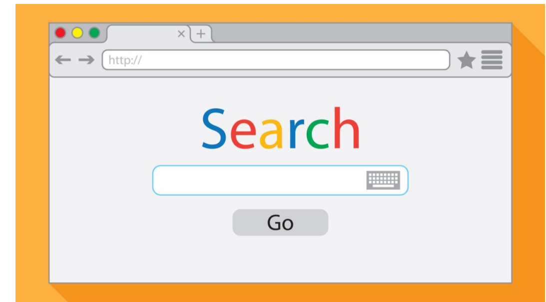
Exercise 6

When would you prefer void Func(int &arg); to void Func(int *arg);? Expand on this distinction for other types besides int.

- When you don't want to deal with pointer semantics, use references
- When you don't want to copy stuff over (doesn't create a copy, especially for parameters and/or return values), use references
- Style wise, we want to use **references for input parameters** and **pointers for output parameters**, with the output parameters declared last

Homework 2 Overview

Homework 2



- Main Idea: Build a search engine for a file system
    - It can **take in queries** and **output a list of files** in a directory that has that query
    - The query will be **ordered** based on the number of times the query is in that file
    - Should handle **multiple word queries** (*Note: all words in a query have to be in the file*)

- What does this mean?
    - Part A: **Parsing a file** and reading all of its contents into heap allocated memory
    - Part B: **Crawling a directory** (reading all regular files recursively in a directory) and building an index to query from
    - Part C: **Build a searchshell** (search engine) to query your index for results

Note: It will use the **LinkedList** and **HashTable** implementations from **HW1**!

Part A: File Parsing

Read a file and generate a HashTable of WordPositions!

Word positions will include the word and LinkedList of its positions in a file.

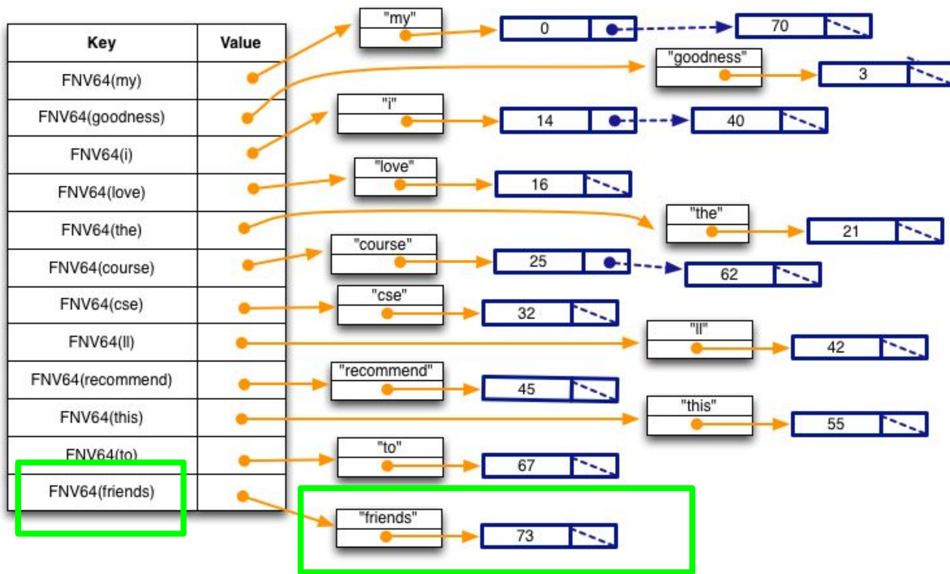
```c
typedef struct WordPositions {
  char        *word;      // normalized word.  Owned.
  LinkedList  *positions; // list of DocPositionOffset_t.
} WordPositions;
```

Note that the key is the hashed C-string of `WordPositions`

somefile.txt

```
My goodness!  I love the course CSE333.\n
I'll recommend this course to my friends.\n
```

`ParseIntoWordPositionsTable(contents)`



Part B: Directory Crawling – DocTable

Read through a directory in CrawlFileTree.c

For each file visited, build your DocTable and MemIndex!

DocTable maps document names to IDs.
FNV64 is a hash function.

```
struct doctable_st {
  HashTable *id_to_name;   // mapping doc id to doc name
  HashTable *name_to_id;   // mapping docname to doc id
  DocID_t    max_id;       // max docID allocated so far
};
DocID_t DocTable_Add(DocTable *table, char *doc_name);
```

Key	Value
5	→ "test_tree/README.TXT"
1	→ "test_tree/books/ulysses.txt"
4	→ "test_tree/bash-4.2/trap.c"
2	→ "test_tree/enron_email/2."
3	→ "test_tree/example.txt"

docid_to_docname

Key	Value
FNV64("test_tree/README.TXT")	→ (DocID_t) 5
FNV64("test_tree/example.txt")	→ (DocID_t) 3
FNV64("test_tree/enron_email/2.")	→ (DocID_t) 2
FNV64("test_tree/bash-4.2/trap.c")	→ (DocID_t) 4
FNV64("test_tree/books/ulysses.txt")	→ (DocID_t) 1

docname_to_docid

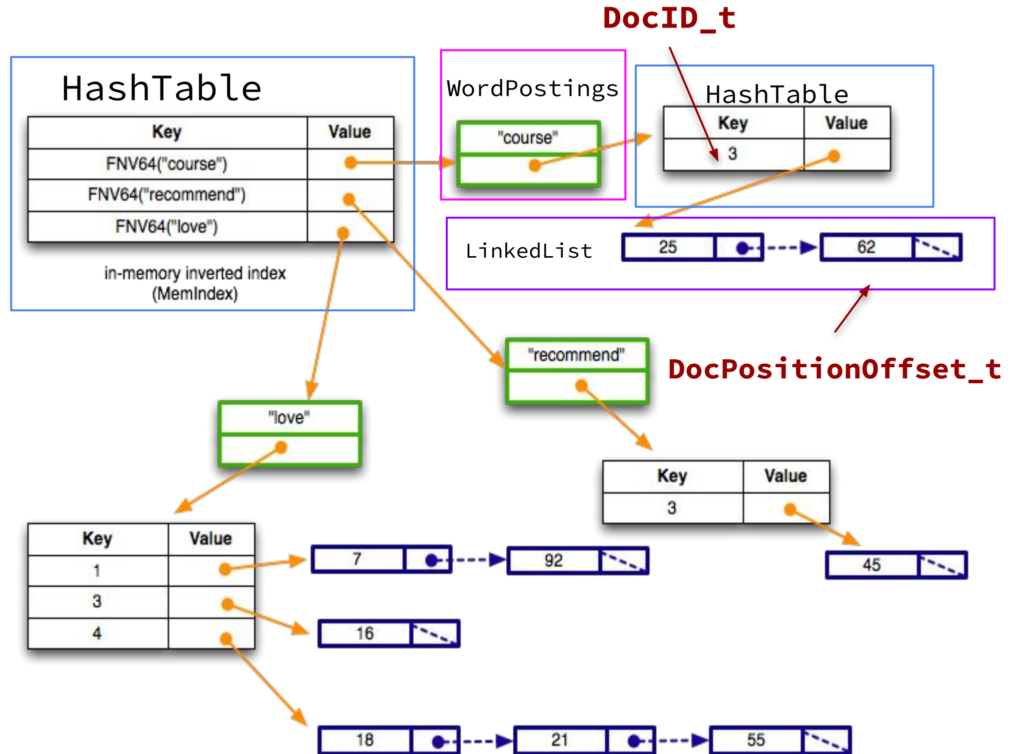
Part B: Directory Crawling – MemIndex

MemIndex is an index to view files.
It's a HashTable of WordPostings.

```
typedef struct {
 char         *word;
 HashTable    *postings;
} WordPostings;
```

Let's try to find what contains "course":

- WordPostings' postings has an element with `key == 3` (Only DocID 3 has "course in its file")
- The value is the LinkedList of offsets the words are in DocID 3



Part C: Searchshell

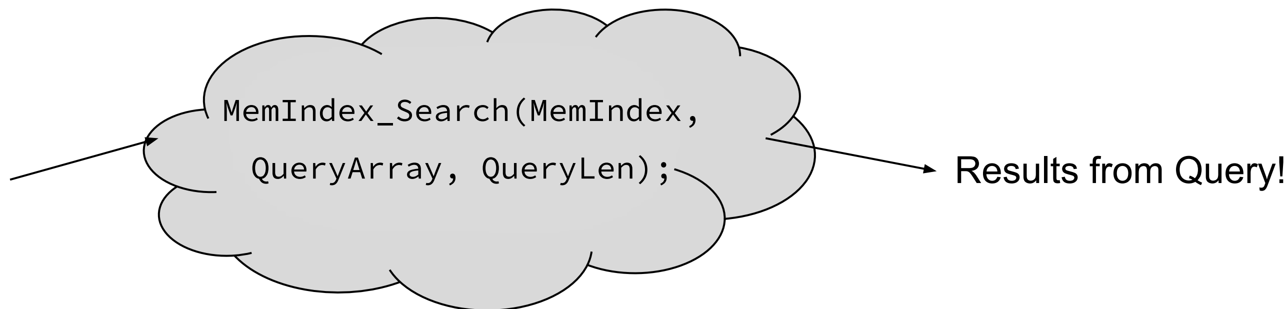
- Use queries to ask for a result!
  - Formatting should match example output
  - Exact implementation is up to you!

MemIndex.h

```
typedef struct SearchResult {
 uint64_t docid;  // a document that matches a search query
 uint32_t rank;   // an indicator of the quality of the match
} SearchResult, *SearchResultPtr;
```

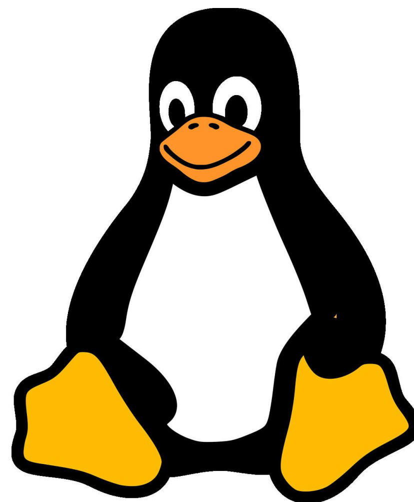
Query

`course friends my`



Hints

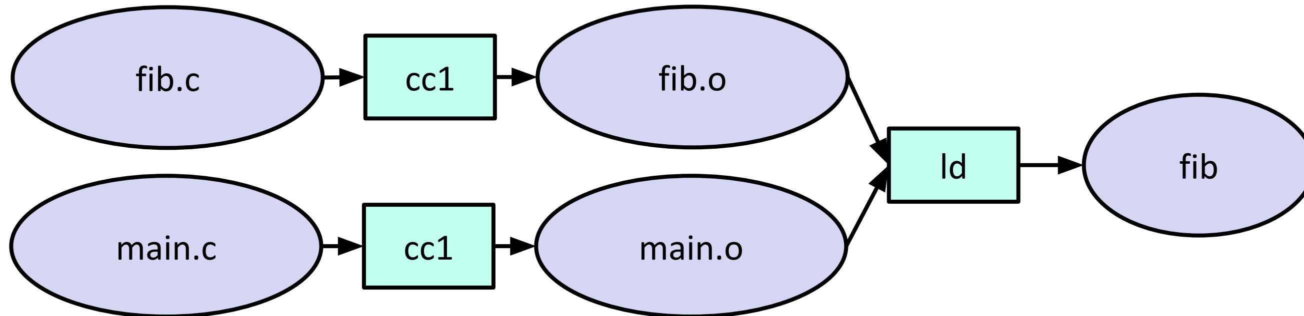
- Read the `.h` files for documentation about functions!
- Understand the high level idea and data structures before getting started
- Follow the suggested implementation steps given in the CSE 333 HW2 spec



Extern and Static

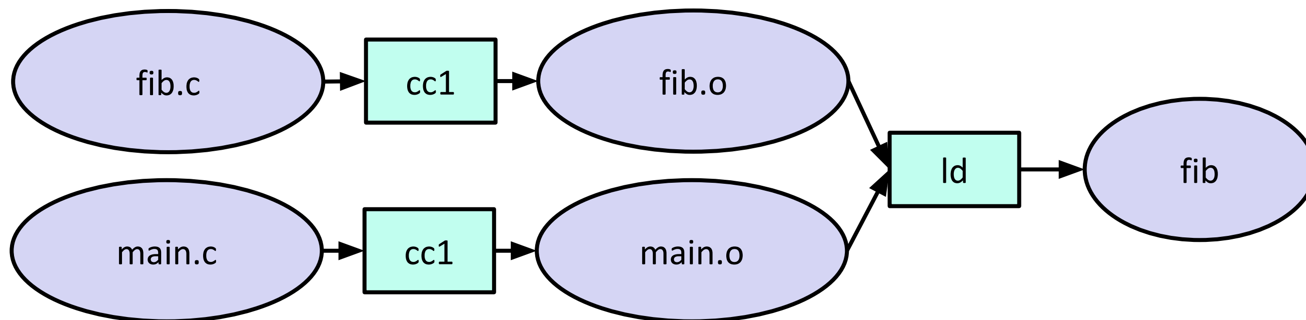
Extern and Static

- `extern` makes a **declaration** visible in any module, but tells the linker to look for the **definition** in a different module
- `static` makes a **definition** private to the current module, and disallows access from other modules *regardless of any further extern declaration*
- `#include`'s make it difficult to reason about which files have the declarations and definitions :(



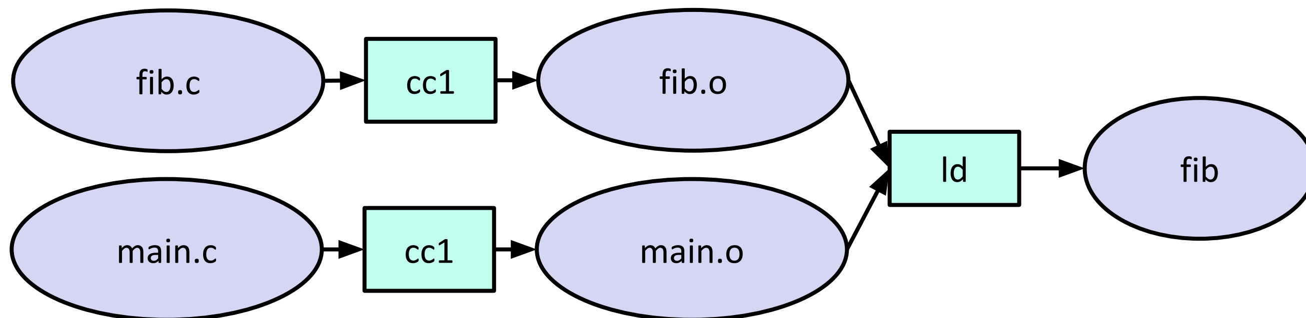
Extern and Static: A Few Examples …

- Scenario 1:
  - We have an **`extern`'ed declaration** in `fib.h`, which is `#include`'d into the `fib` and `main` modules
  - There is nothing in `fib.c`



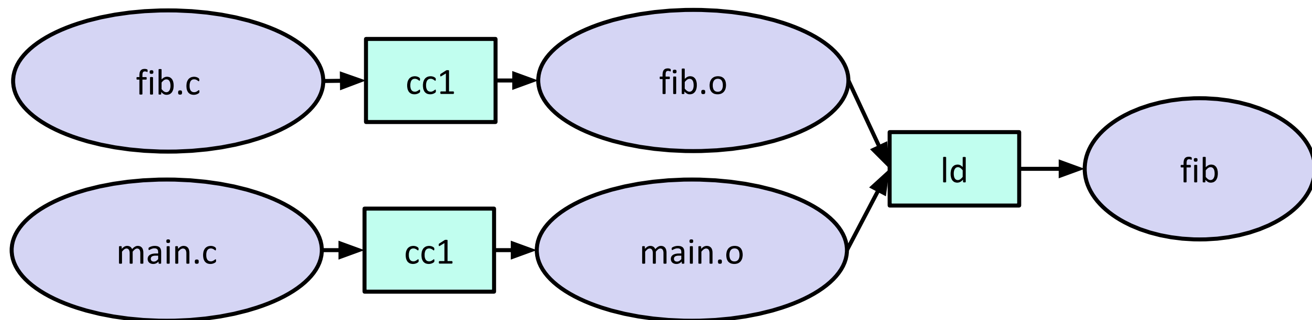
Extern and Static: A Few Examples …

- Scenario 2:
  - We have an **`extern`'ed declaration** in `fib.h`, which is `#include`'d into the `fib` and `main` modules
  - There is a definition in `fib.c`



Extern and Static: A Few Examples …

- Scenario 3:
  - We have a **`static`'ed definition** in `fib.h`, which is `#include`'d into the `fib` and `main` modules
  - We remove the definition from `fib.c`



Extern and Static: A Few Examples …

- Scenario 4:
  - We have no declarations nor definitions in `fib.h`, which continues to be `#include`'d into the `fib` and `main` modules
  - We put the definition back into `fib.c`