CSE 333
Section 4

Makefiles, C++ Intro, HW2 Overview

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 (7/21) @ 10:00 am

Exercise 10: Due Wednesday (7/23) @ 10:00 am

Homework 2: Due Thursday (7/24) @ 11:00 pm

Makefile Demo

`make`

- `make` is a classic program for controlling what gets (re)compiled and how
  - Many other such programs exist (*e.g.* `ant`, `maven`, IDE "projects")

- `make` has tons of fancy features, but only two basic ideas:
  1) Scripts for executing commands
  2) Dependencies for avoiding unnecessary work

- To avoid "just teaching `make` features" (boring and narrow), let's focus more on the concepts…

Building Software

- Programmers spend a lot of time “building”
  - Creating programs from source code
  - Both programs that they write and other people write
- Programmers like to automate repetitive tasks
  - Repetitive: gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c
    - Retype this every time: 😭
    - Use up-arrow or history: 😐 (still retype after logout)
    - Have an alias or bash script: 🙂
    - Have a Makefile: 😊 (you’re ahead of us)

"Real" Build Process

- On larger projects, you don't want to have one big (set of) command(s) that redoes everything on every change:
  1) If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
  2) If source files have multiple outputs (*e.g.* javadoc), you'd have to type out the source file name(s) multiple times
  3) You don't want to have to document the build logic when you distribute source code
  4) You don't want to recompile everything every time you change something (especially if you have 10^5-10^7 files of source code)
- A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

An Example

- We have a small program that is split into multiple tiny modules (code on the web linked to this lecture):

`main.c` `speak.h` `speak.c` `shout.h` `shout.c`

- Modules:
  - speak.h/speak.c: write a string to stdout
  - shout.h/shout.c: write a string to stdout LOUDLY
  - main.c: client program
- Demo: build this program incrementally, and recompile only necessary parts when something changes
- How do we automate this “minimal rebuild”?

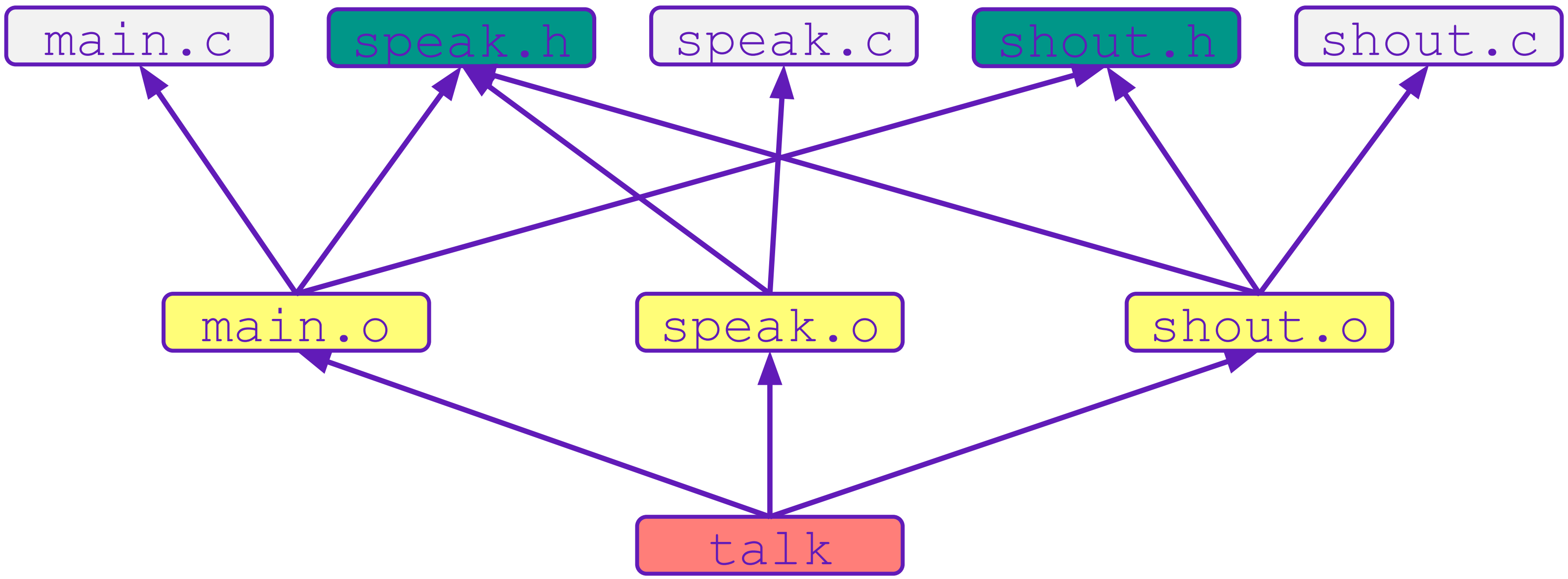
Recompilation Management

- The "theory" behind avoiding unnecessary compilation is a *dependency dag* (**d**irected, **a**cyclic **g**raph)

- To create a target t, you need sources $s_1, s_2, \ldots, s_n$ and a command c that directly or indirectly uses the sources
  - It t is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source s_i is itself a target for some other sources, see if it needs to be rebuilt…
  - Cycles "make no sense"!

Theory Applied to Our Example

- What are the dependencies between built and source files?
- What needs to be rebuilt if something changes?



`make` Basics

- A makefile contains a bunch of triples:

```
target: sources
← Tab → command
```

  - Colon after target is *required*
  - Command lines must start with a **TAB**, NOT SPACES
  - Multiple commands for same target are executed *in order*
    - Can split commands over multiple lines by ending lines with '\'

- Example:

```
foo.o: foo.c foo.h bar.h
    gcc -Wall -o foo.o -c foo.c
```

- Demo: look at Makefile for our example program

Using `make`

```
bash% make -f <makefileName> target
```

- Defaults:
  - If no `-f` specified, use a file named `Makefile`
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell
    - Set `SHELL` variable in makefile to ensure
- Target execution:
  - Check each source in the source list:
    - If the source is a target in the Makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

make Variables

- You can define variables in a makefile:
  - All values are strings of text, no “types”
  - Variable names are case-sensitive and can’t contain ‘:’, ‘#’, ‘=’, or whitespace
- Example:

```
CC = gcc
CFLAGS = -Wall -std=c17
foo.o: foo.c foo.h bar.h
    $(CC) $(CFLAGS) -o foo.o -c foo.c
```

- Advantages:
  - Easy to change things (especially in multiple commands)
  - Can also specify on the command line (`CC=clang FLAGS=-g`)

More Variables; "phony" targets

(2 separate things)

- It's common to use variables to hold list of filenames:

```
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
    gcc -o widget $(OBJFILES)
clean:
    rm $(OBJFILES) widget *~
```

- `clean` is a convention
  - Remove generated files to "start over" from just the source
  - It's "funny" because the target doesn't exist and there are no sources, but it works because:
    - The target doesn't exist, so it must be "remade" by running the command
    - These "phony" targets have several uses, such as "`all`"…

“all” Example

```
all: prog B.class someLib.a
    # notice no commands this time

prog: foo.o bar.o main.o
    gcc -o prog foo.o bar.o main.o

B.class: B.java
    javac B.java

someLib.a: foo.o baz.o
    ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
    gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

Revenge of the Funny Characters

- Special variables:
  - `$@` for target name
  - `$^` for all sources
  - `$<` for left-most source
  - Lots more! – see the documentation

- Examples:

```
# CC and CFLAGS defined above
widget: foo.o bar.o
    $(CC) $(CFLAGS) -o $@ $^
foo.o: foo.c foo.h bar.h
    $(CC) $(CFLAGS) -c $<
```

And more…

- There are a lot of "built-in" rules – see documentation
- There are "suffix" rules and "pattern" rules
  - Example:

```
%.class: %.java
    javac $<   # we need the $< here
```

- Remember that you can put *any* shell command – even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day…)

Exercise 1

Exercise 1: File DAG

We have the following files:

Point.h
```
class Point { … };
```

UsePoint.cc
```
#include "Point.h"
#include "Thing.h"
int main( … ) { … }
```

UseThing.cc
```
#include "Thing.h"
int main( … ) { … }
```

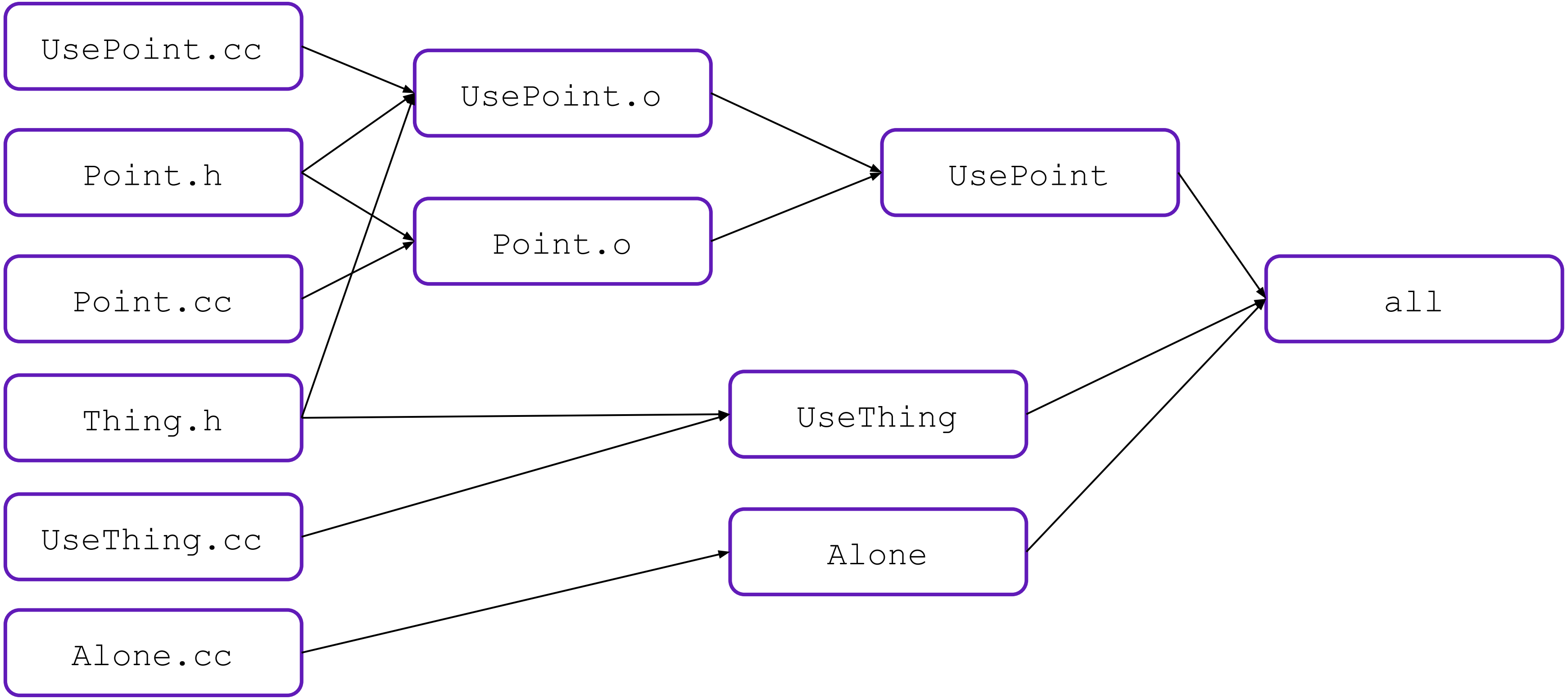
Point.cc
```
#include "Point.h"
// defs of methods
```

Thing.h
```
struct Thing { … };
// full struct def here
```

Alone.cc
```
int main( … ) { … }
```

Draw a DAG (directed acyclic graph) to represent the dependencies between source files and targets.

Exercise 1: File DAG



Exercise 1: Makefile

Write the corresponding Makefile for Point.

```
CFLAGS = -Wall -g -std=c++17

all: UsePoint UseThing Alone

UsePoint: UsePoint.o Point.o
	g++ $(CFLAGS) -o UsePoint UsePoint.o Point.o

UsePoint.o: UsePoint.cc Point.h Thing.h
	g++ $(CFLAGS) -c UsePoint.cc

Point.o: Point.cc Point.h
	g++ $(CFLAGS) -c Point.cc

UseThing: UseThing.cc Thing.h
	g++ $(CFLAGS) -o UseThing UseThing.cc

Alone: Alone.cc
	g++ $(CFLAGS) -o Alone Alone.cc

clean:
	rm UsePoint UseThing Alone *.o *~
```

Pointers, References, & Const

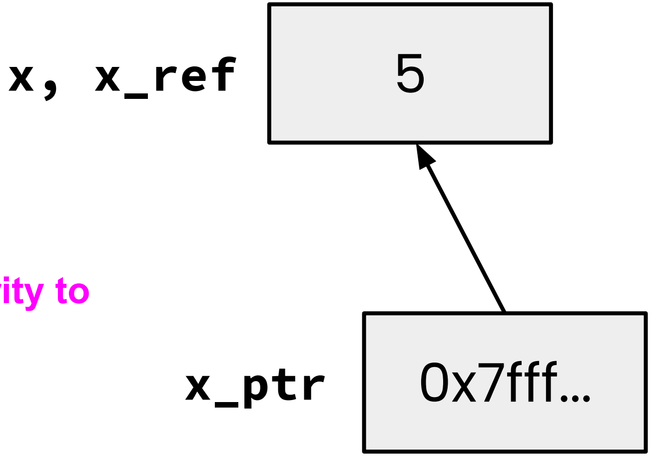
Example

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`

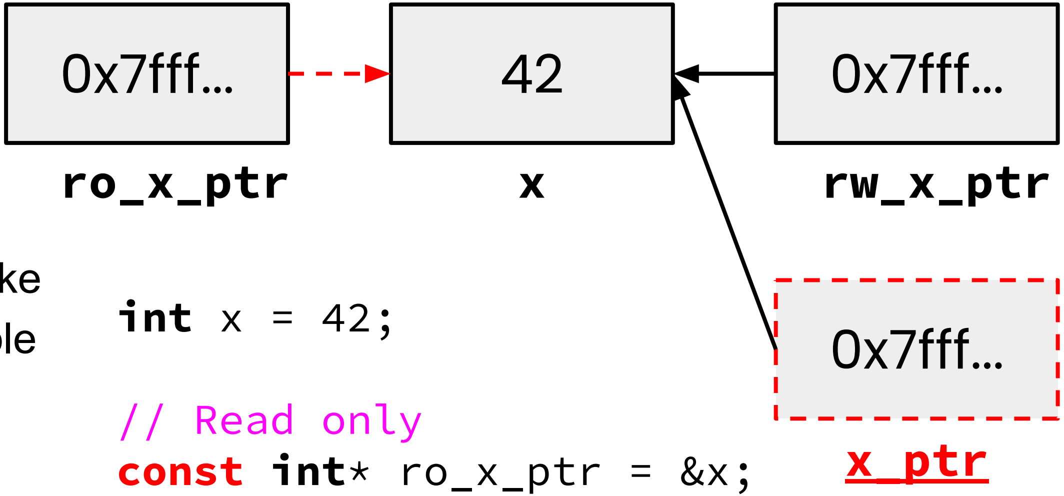
References

- 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
- <u>Does not change the underlying write-permissions</u> for this variable

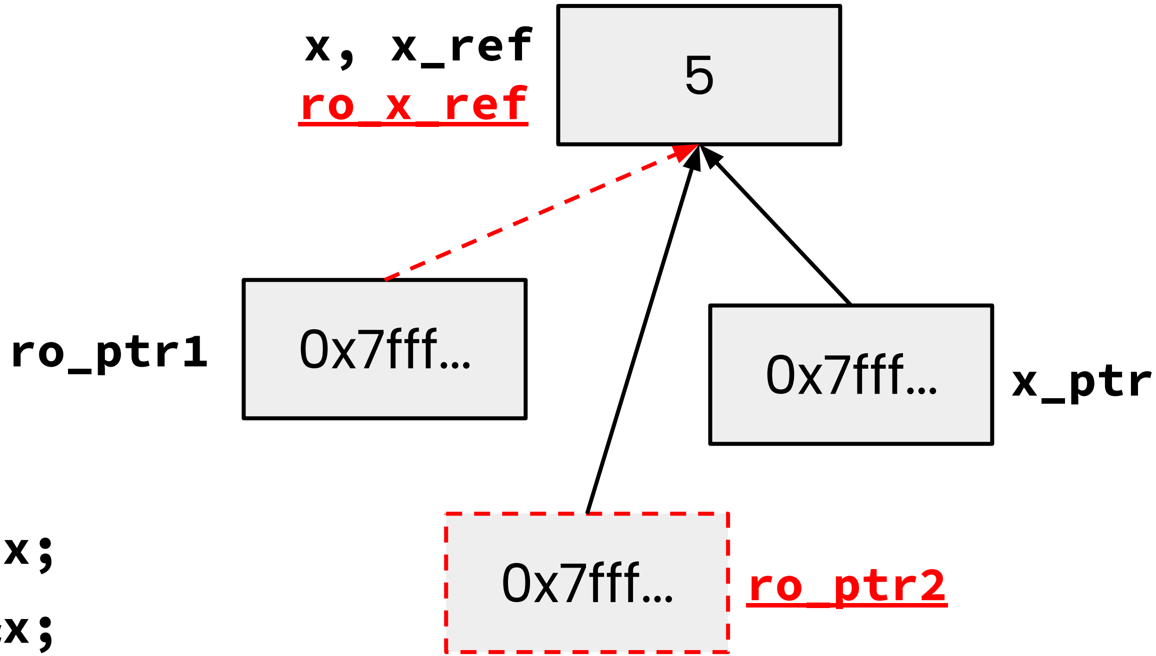
```
int x = 42;

// 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;
```

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

Exercise 2

Exercise 2

```
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"

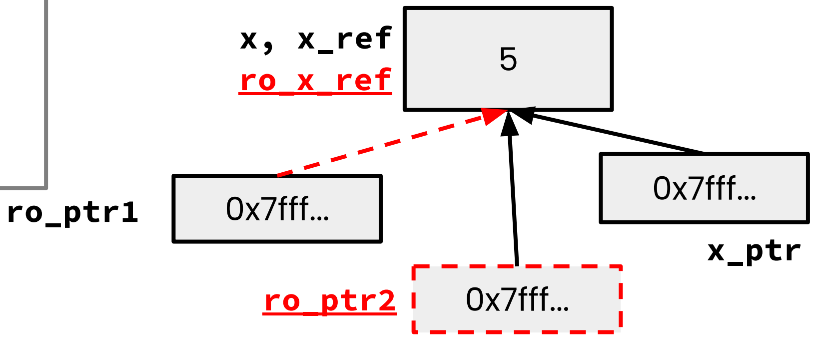
Tip: Read the declaration "right-to-left"

Legend

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

Exercise 2

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

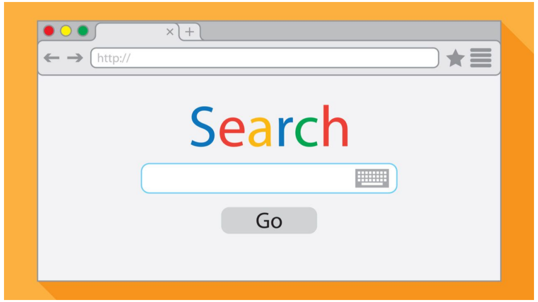
Exercise 2

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

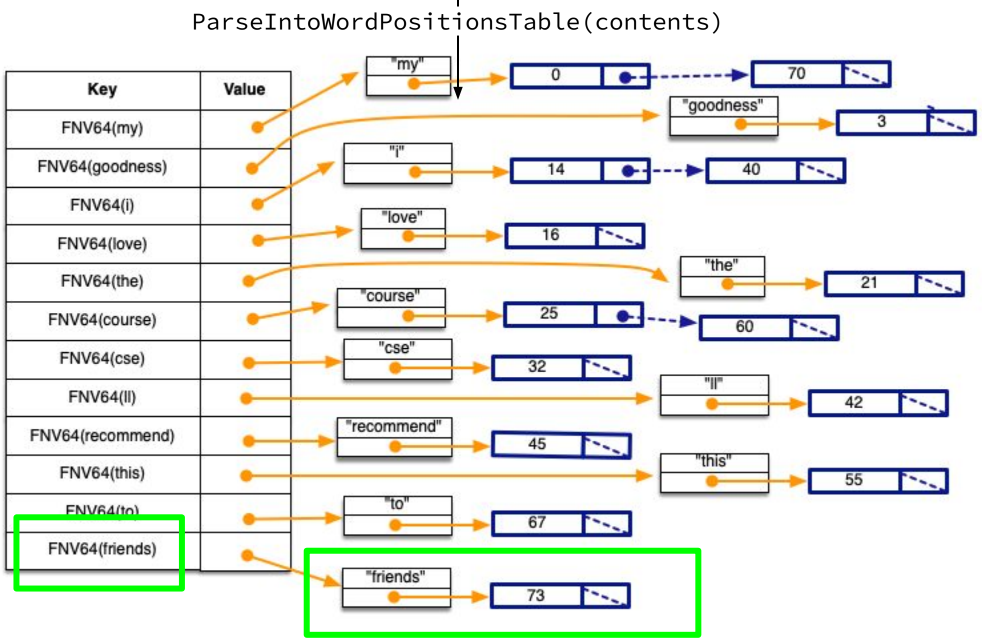
Read a file and generate a HashTable of WordPositions!

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

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

somefile.txt

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



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

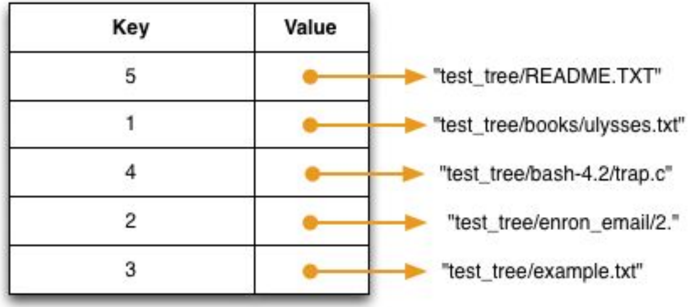
Part B: Directory Crawling – DocTable

Read through a directory in CrawlFileTree.c

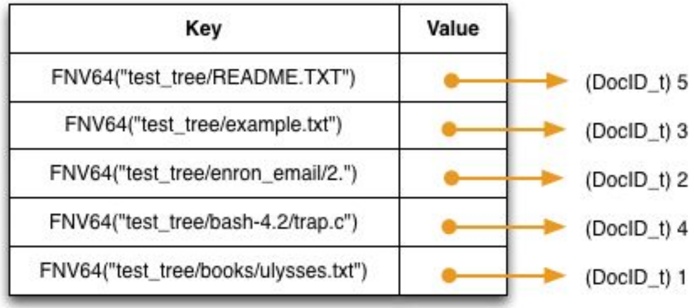
For each file visited, build your DocTable and MemIndex!

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);
```



docid_to_docname



docname_to_docid

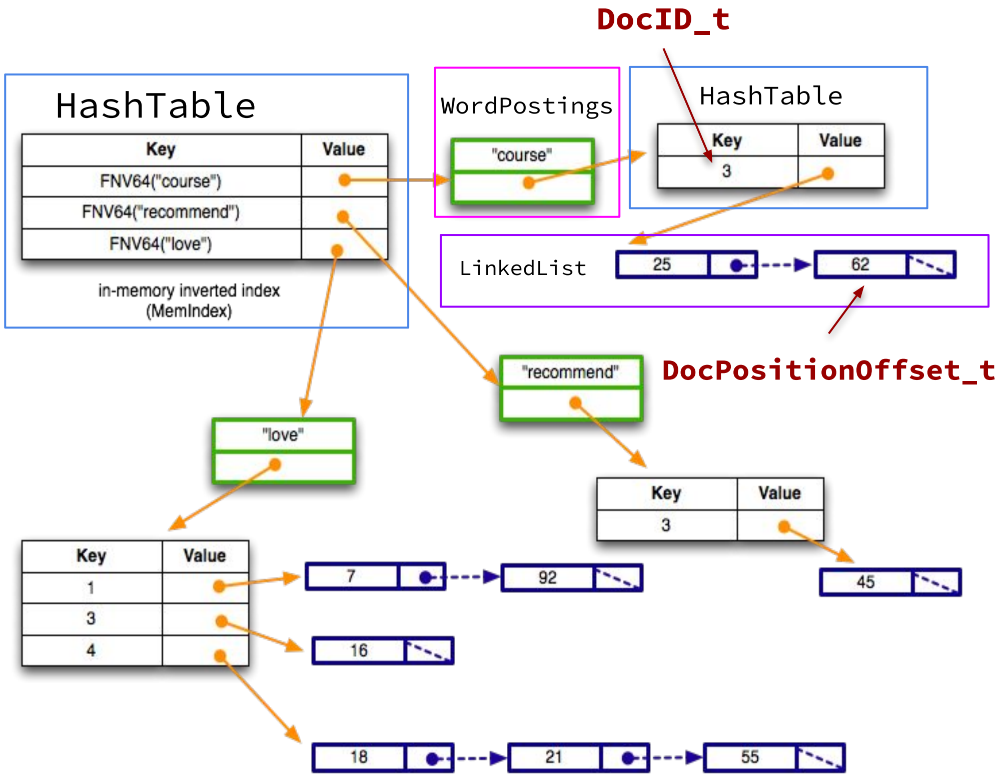
Part B: Directory Crawling – MemIndex

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

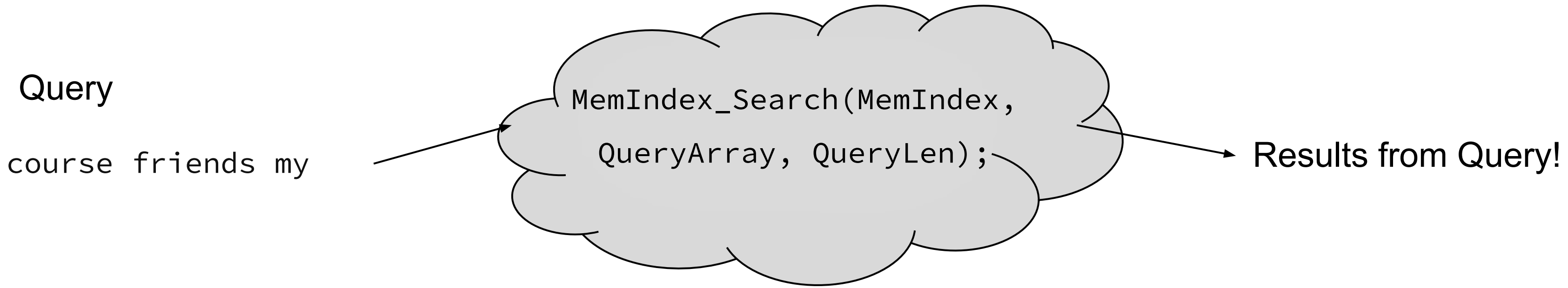


Part C: Searchshell

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

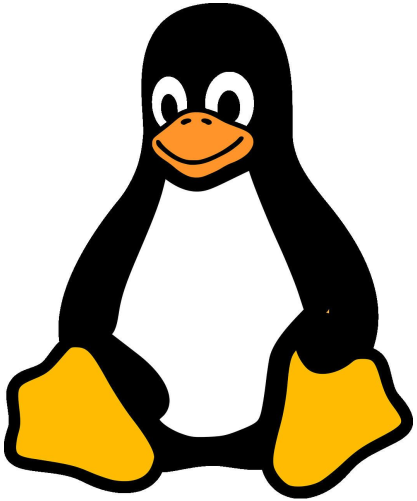
MemIndex.h

```c
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;
```



Hints

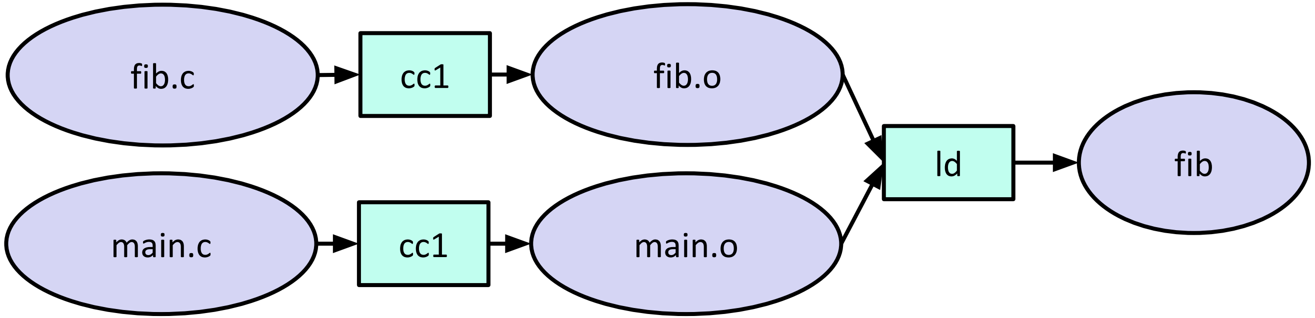
- Read the `.h` files for documentation about functions!
- Understand the high level idea and data structures before getting started
- Follow the suggested implementation steps given in the CSE 333 HW2 spec



Extern and Static

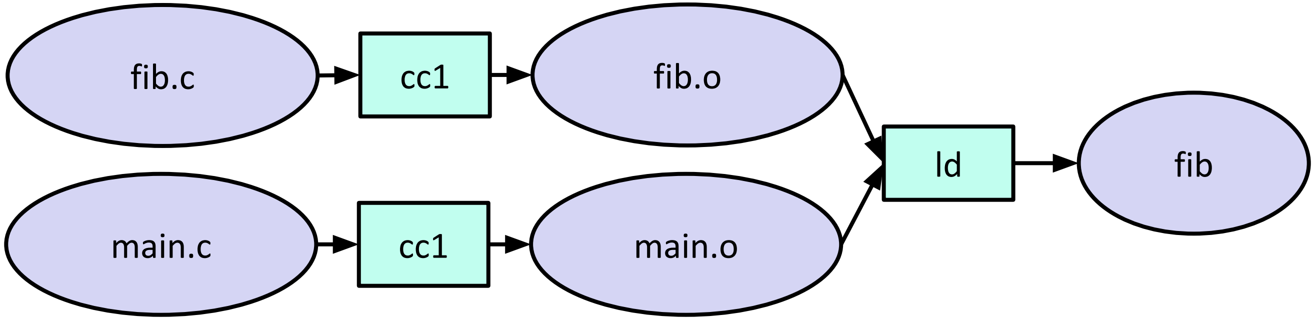
Extern and Static

- `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 :(



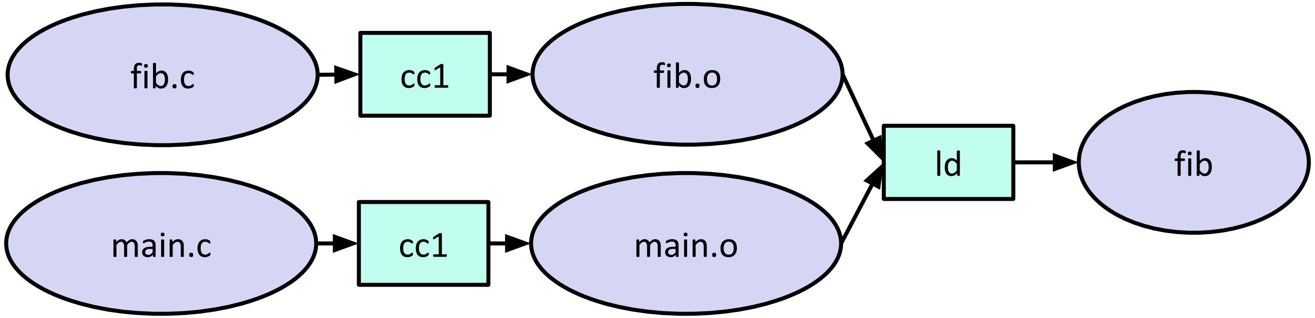
Extern and Static: A Few Examples …

- 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`



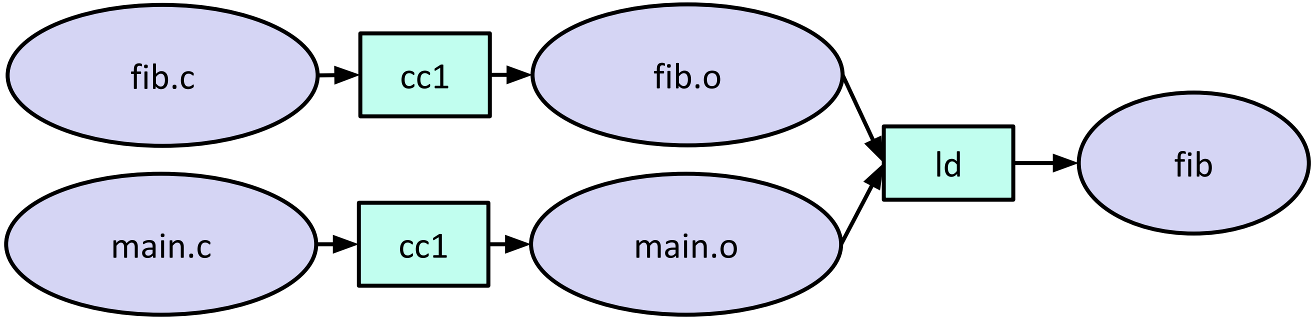
Extern and Static: A Few Examples …

- 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`