CSE 333
Section 4
HW2 Overview, C++ Intro
Logistics

● Exercise 5
  ○ Due Thursday (10/19) by 10pm

● Midterm
  ○ Next Friday (10/27) 11:30 - 12:20 pm @ Kane 110

● Homework 2
  ○ Due Monday (10/30) @ 10:00 pm
  ○ Indexing files to allow for searching
  ○ Bigger and longer than Homework 1!
Agenda

- Makefiles
- HW2 overview
- C++ Intro review
- Exercise 1
- Objects and const methods review
- Exercise 2
Makefiles

target: src1 src2 ... srcN
command/commands

Makefiles are used to manage project recompilation. Project structure / dependencies can be represented as a DAG, which a Makefile encodes to recursively build the minimum number of files for a target.
Makefiles

- In practice, these can often be written automatically or by following common target patterns
  - In 333, we will ask you to submit Makefiles along with a few of your exercises, but you can adapt existing rules from provided examples
  - It is more important that you understand the concepts behind them and can read and understand target rules from a given Makefile

- Exercise 3 on your worksheet is provided for practice on your own time; solutions will be released with the rest of the worksheet solutions
Homework 2 Overview
Homework 2 Overview

- Build a search engine for a collection of files
  - User inputs a text query (one or more words)
  - The search engine outputs a ranked list of files (decreasing order) within the collection that match the query
Homework 2 Overview

● Build a search engine for a collection of files
  ○ User inputs a text query (one or more words)
  ○ The search engine outputs a ranked list of files (decreasing order) within the collection that match the query

● More details:
  ○ Our collection of files will be the contents of a specified local directory (including the contents of its subdirectories)
  ○ Naive matching: any file that contains all words in the query
  ○ Naive ranking: sum of the counts of all words in the query
    ■ Files in search results with equal ranking can be displayed in any order
Search Engine Implementation Overview

- Major components:
  - The directory crawler recursively finds the “regular” files in the specified collection/corpus
Search Engine Implementation Overview

- Major components:
  - The **directory crawler** recursively finds the “regular” files in the specified collection/corpus
  - As files are found, the **file parser** adds the words and their locations into heap-allocated data structures
    - This uses the **LinkedList** and **HashTable** implementations from HW1 – need **libhw1.a** to be in the **hw1/ directory**
Search Engine Implementation Overview

- Major components:
  - The **directory crawler** recursively finds the “regular” files in the specified collection/corpus
  - As files are found, the **file parser** adds the words and their locations into heap-allocated data structures
    - This uses the LinkedList and HashTable implementations from HW1 – need libhw1.a to be in the hw1/ directory
  - The **searchshell** (i.e., search engine) reads in user queries and uses the built up data structures to return the search results
    - Finish the infinite loop by using Ctrl-D
Part A: File Parsing

Read a file and generate a HashTable of `WordPositions`

- The words are “normalized” – lowercase and broken by non-alphabetic characters
- HashTable key is the hashed normalized word
- `WordPositions` has heap-allocated copy of the word and a LinkedList of its position(s) in the file.

def ParseIntoWordPositionsTable(contents):
    typedef struct {
        char *word; // in heap (owned)
        LinkedList *positions; // DocPositionOffset_t
    } WordPositions;
Part B: Directory Crawling – DocTable

Recursively search directories and parse files to build out a DocTable and MemIndex for the collection of files

- **DocTable** maps document names to IDs (in both directions) via HashTables

```c
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);
```
Part B: Directory Crawling – MemIndex

- MemIndex indexes individual words to their locations in the collection of files via a HashTable of WordPostings.

Let's examine the word “course”:

- The WordPostings’ HashTable has single key, so only DocID/file 3 contains “course”
- The LinkedList shows it appears at characters 25 and 62 in DocID 3

```c
typedef struct {
    char     *word;
    HashTable *postings;
} WordPostings;
```
Part C: Searchshell

Parse user queries, use MemIndex to generate search results, then output to list with ranks

- Formatting should match example output, other than ordering of ties
- Fairly open-ended – the exact implementation is up to you!

```c
typedef struct SearchResult {
  uint64_t docid; // matching document
  uint32_t rank; // rank quantifier
} SearchResult;
```

```
MemIndex_Search(MemIndex, QueryArray, QueryLen);
```

```
Query: course friends
```

```
my
```

```
LinkedList of SearchResult
```

```
typedef struct SearchResult {
  uint64_t docid; // matching document
  uint32_t rank; // rank quantifier
} SearchResult;
```
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 HW2 spec
- Debug on very small sets of short text files
  - You can create your own directory and files!
C++ introduction review: Pointers, References, & Const
Example

Consider the following code:

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

When would it be a good idea to use references instead of pointers?

Still the address-of operator!

Note syntactic similarity to pointer declaration!
Pointers vs. References

**Pointers**
- Can move to different data via reassignment/pointer arithmetic
- Can be initialized to `nullptr`
- 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/nullptr
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.

```c
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;
```
Exercise 1
Exercise 1

```
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;
```

Tip: Read the declaration “right-to-left”

Legend
Red = can’t change box it’s next to
Black = read and write
Exercise 1

```c
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?

- ✔ Bar(x_ref);
- ✔ Foo(x_ref);
- ✔ ro_ptr1 = (int*) 0xDEADBEEF;
- ✔ x_ptr = &ro_x_ref;
- ✔ x_ptr = ro_ptr2 + 2;
- ✗ *ro_ptr1 = *ro_ptr1 + 1; (*ro_ptr1) is const

Legend:
- Red = can't change box it's next to
- Black = "read and write"
Exercise 1

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
Objects and const Methods
```cpp
#ifndef POINT_H_
#define POINT_H_

class Point {
  public:
    Point(const int x, const int y);
    int get_x() const { return x_; }
    int get_y() const { return y_; }
    double Distance(const Point& p) const;
    void SetLocation(const int& x, const int& y);
  
  private:
    int x_;  
    int y_; 
};  // class Point

#endif // POINT_H_
```

**Cannot** mutate the object it’s called on. Trying to change `x_` or `y_` inside will produce a compiler error!

A **const** class object can only call member functions that have been declared as **const**.
Exercise 2
class MultChoice {
  public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { } // 2-arg ctor
    int get_q() const { return q_; }
    char getResp() { return resp_; }
    bool Compare(MultChoice &mc) const; // do these MultChoice's match?
  private:
    int q_;       // question number
    char resp_;   // response: 'A','B','C','D', or 'E'
}; // class MultChoice

const MultChoice m1(1,'A');
MultChoice m2(2,'B');
// OK
cout << m1.getResp();
// OK
cout << m2.get_q();
// ERROR
m1.Compare(m2);
// OK
m2.Compare(m1);
What would you change about the class declaration to make it better?

class MultChoice {
public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { } // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() { return resp_; }
    bool Compare(MultChoice &mc) const; // do these MultChoice's match?

private:
    int q_;    // question number
    char resp_; // response: 'A','B','C','D', or 'E'
}; // class MultChoice
class MultChoice {
public:
    MultChoice(int q, char resp) : q_(q), resp_(resp) { } // 2-arg ctor
    int get_q() const { return q_; }
    char get_resp() { return resp_; }
    bool Compare(MultChoice &mc) const; // do these MultChoice's match?

private:
    int q_; // question number
    char resp_; // response: 'A','B','C','D', or 'E'
}; // class MultChoice

- make get_resp() const
- make the parameter to Compare() const
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!*
Exercise 3a

Which *lines* of the snippets of code below would cause compiler errors?

- ✔ int z = 5;
- ✔ const int* x = &z;
- ✔ int* y = &z;
- ✔ x = y;
- ✔ *x = *y;
- ❌

---

- ✔ int z = 5;
- ✔ int* const w = &z;
- ✔ const int* const v = &z;
- ❌ *v = *w;
- ❌ *w = *v;
## Exercise 3

### Point.h
```cpp
class Point {
    ...
};
```

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

### Point.cc
```cpp
#include "Point.h"

// defs of methods
```

### Thing.h
```cpp
struct Thing {
    ...
};

// full struct def here
```

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

### Alone.cc
```cpp
int main(...)
{
    ...
}
```

1. **Draw out Point’s DAG**
   - The direction of the arrows is not important, but be consistent

https://courses.cs.washington.edu/courses/cse333/23wi/lectures/07/07-syscalls-make_23wi.pdf#page=37
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 *~
Q&A :-)
