CSE 333 Section 4

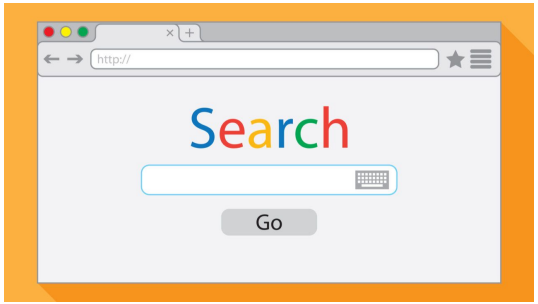
HW2, C++ Intro, `ref`, `const`, Classes

Logistics

- Homework 2
  - Due next **Thursday (7/21) @ 11:59pm**
  - Indexing files to allow for searching
- Exercise 5
  - Due **Friday (7/15) @ 11:00am**
- Exercise 6
  - Due next **Monday (7/18) @ 11:00am**

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`!

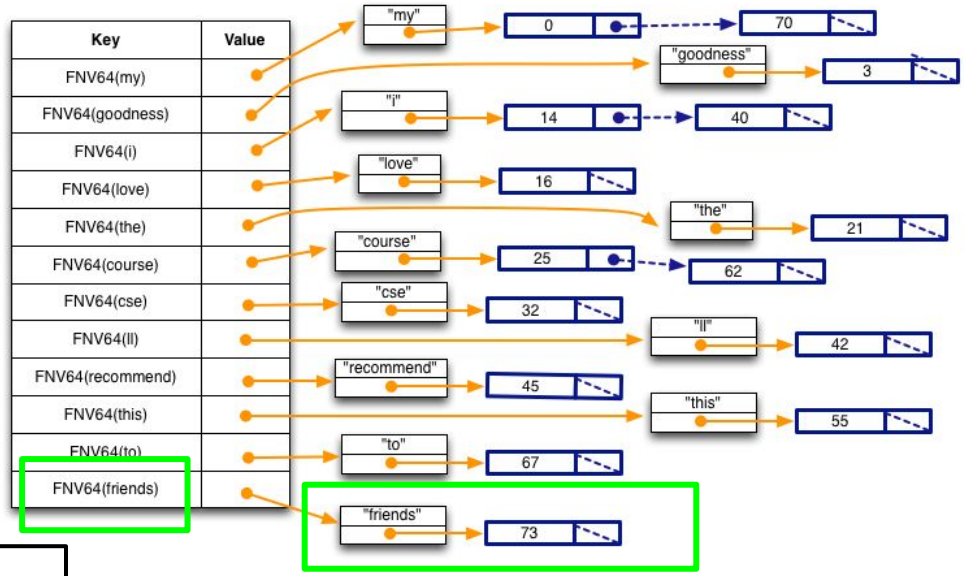
Word positions will include the word and `LinkedList` of its positions in a file.

```
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
```

`ParseIntoWordPositionsTable(contents)`



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

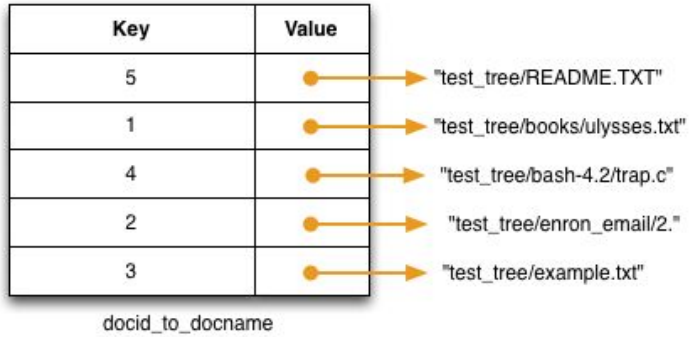
Part B: Directory Crawling – DocTable

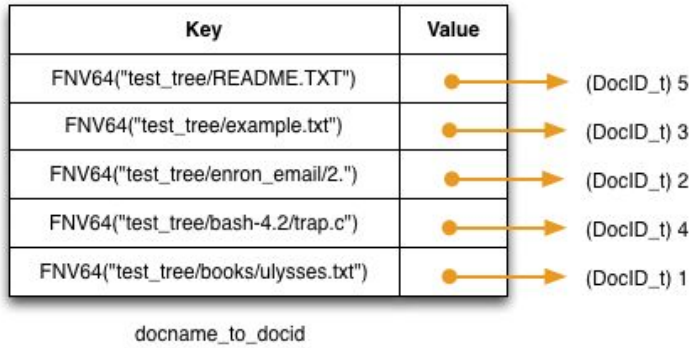
Read through a directory in `CrawlFileTree.c`

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);
```





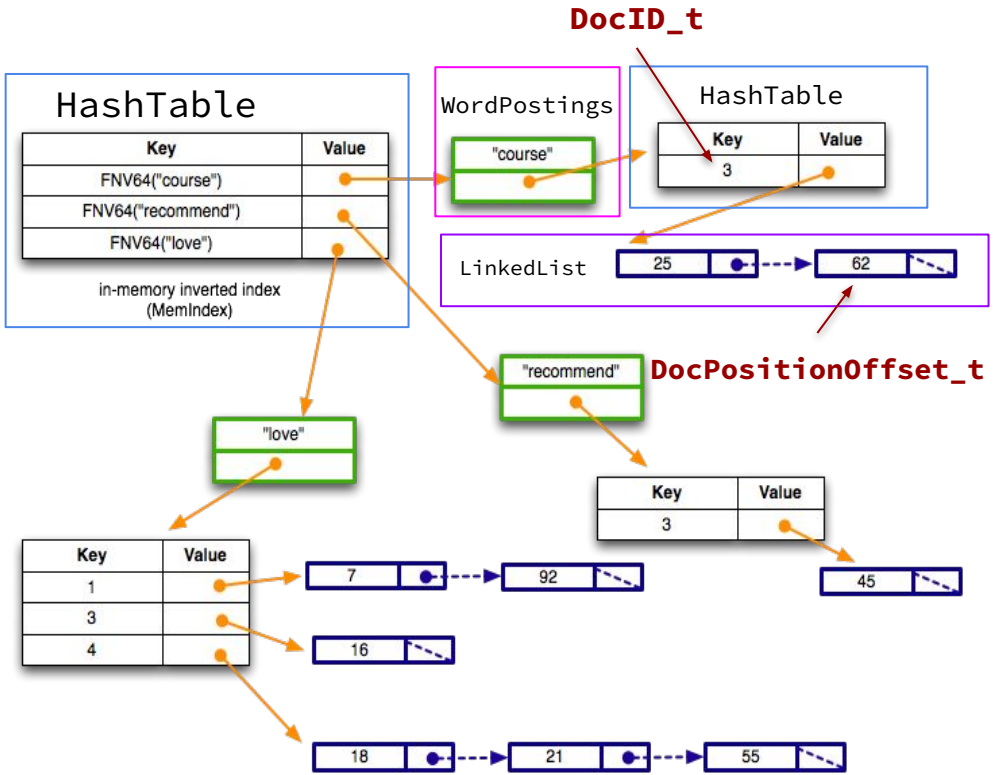
Part B: Directory Crawling – MemIndex

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



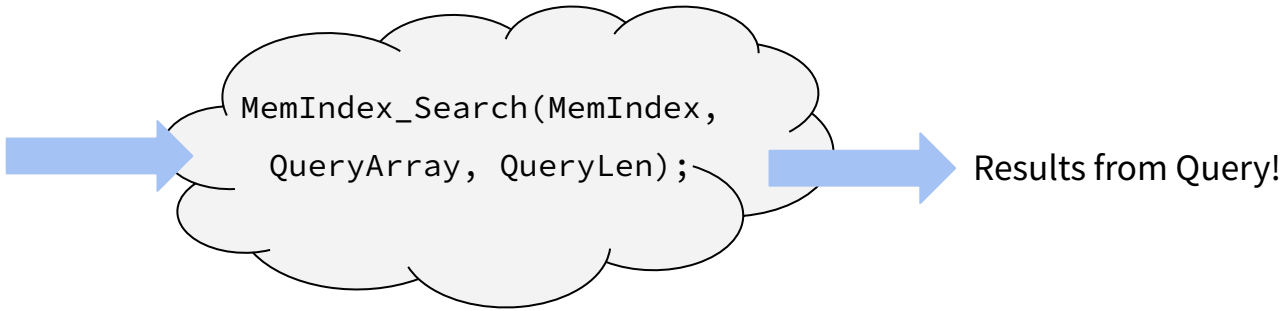
Part C: Searchshell

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

MemIndex.h

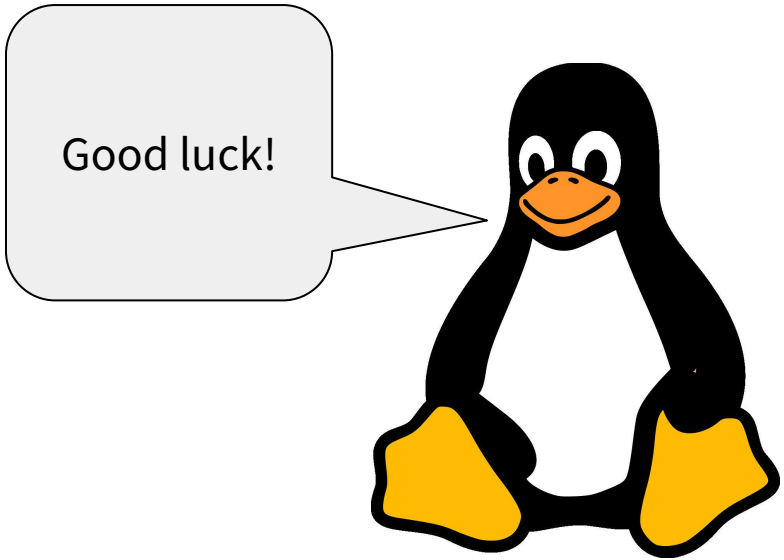
```
typedef struct SearchResult {
 uint64_t docid;  // a document that matches a search query
 uint32_t rank;   // an indicator of the quality of the match
} SearchResult, *SearchResultPtr;
```

<u>Query</u>
`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



Pointers, References, & Const

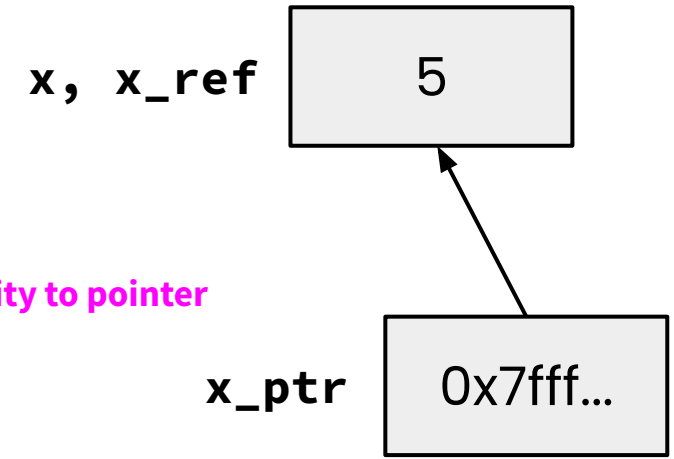
Example

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

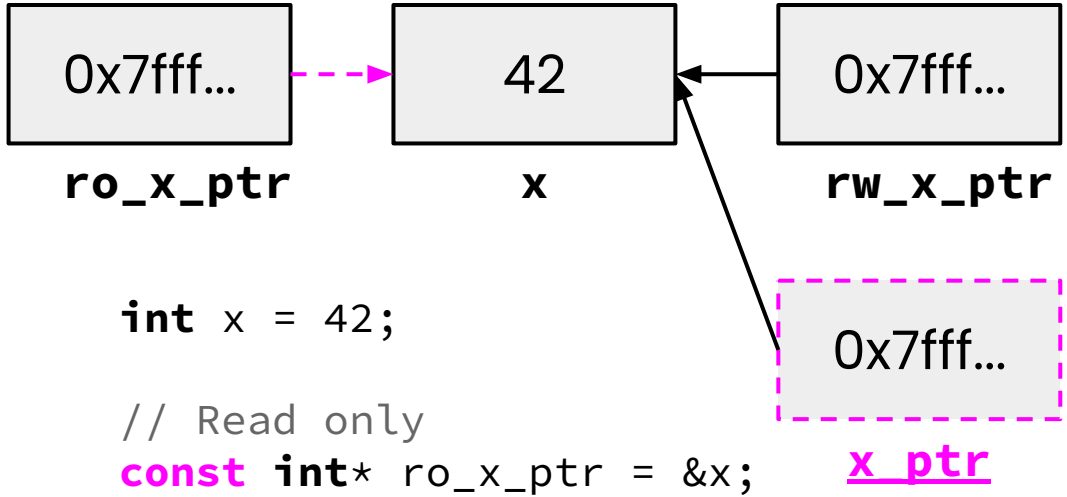
- References the same data for its entire lifetime - *can't reassign*
- No sensible "default reference," must be an alias
- Useful for input parameters:
  `const MyClass& input`

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
  - Can still change x directly




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

Exercise 1

Exercise 1

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

✔️ OK ❌ ERROR

✔️ `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;`

Objects and `const` Methods

```
#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 throw a compiler error!

A **const** class object can only call member functions that have been declared as **const**

Exercise 2

Exercise 2

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

✔️ OK ❌ ERROR

```
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
```

✔️ `const MultChoice m1(1,'A');`
✔️ `MultChoice m2(2,'B');`
❌ `cout << m1.get_resp();`
✔️ `cout << m2.get_q();`

✔️ `const MultChoice m1(1,'A');`
✔️ `MultChoice m2(2,'B');`
✔️ `m1.Compare(m2);`
❌ `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
```

C++ Classes

Review Questions

- What do the following access modifiers mean?

  `public:` Member is accessible by anyone

  `protected:` Member is accessible by this class and any derived classes

  `private:` Member is only accessible by this class

- What is the default access modifier for a `struct` in C++?

  A `struct` can be thought of as a class where all members are default `public` instead of default `private`. In C++, it is also possible to give member functions (such as a constructor) to a `struct`

Constructors Revisited

```
class Int {
 public:
   Int() { ival_ = 17; cout << "default(" << ival_ << ")" << endl; }
   Int(int n) { ival_ = n; cout << "ctor(" << ival_ << ")" << endl; }
   Int(const Int& n) {
     ival_ = n.ival_;
     cout << "cctor(" << ival_ << ")" << endl;
   }
   ~Int() { cout << "dtor(" << ival_ << ")" << endl; }
   . . .
```

Constructor (ctor): Can define any number as long as they have different parameters. Constructs a new instance of the class.

Copy Constructor (cctor): Creates a new instance based on another instance (must take a reference!). Invoked when passing/returning a **non-reference** object to/from a function.

Destructor (dtor): Cleans up the class instance. Deletes dynamically allocated memory (if any).

Initialization Lists

```
MyInt(int x) { ival_ = x; } → MyInt(int x) : ival(x) { }
```

- When is the initialization list of a constructor run, and in what order are data members initialized?

  The initialization list is run before the body of the ctor, and data members are initialized in the order that they are defined in the class, not by initialization list ordering.

- What happens if data members are not included in the initialization list?

  Data members that don't appear in the initialization list are *default initialized/constructed* before ctor body is executed.

Exercise 3

Exercise 3

const omitted from parameters for space

Write a **`Circle`** class with three **`float`** fields: the **x** and **y** coordinates of a **`Circle`**'s center, and its **`radius`**.
Fill in the constructors using initialization lists, then implement the **`get`** and **`set`** functions.

Circle.cc

```
#include "Circle.h"

void Circle::SetCenter(float x, float y) {


}

void Circle::ScaleCircle(float scale) {


}
```

Circle.h

```
class Circle {
 public:
  // Default constructor
  // creates a circle of radius 1 at (0,0)
  _____ { }
  // Parameterized constructor
  // Takes in the x, y coords of the center and a radius
  _____ { }
  // Copy constructor
  // Takes in a reference to another circle
  _____ { }
  // Getters
  float get_x() const { _____ }
  float get_y() const { _____ }
  float get_radius() const { _____ }

  // Setters
  // Sets the center of the circle to the given x and y coords
  void SetCenter(float x, float y);
  // Scales the radius of the circle by the given scale factor
  // For simplicity, negative scale factors are allowed.
  void ScaleCircle(float scale);

 private:
  // the location of the center of a circle and its radius

};  // class Circle
```

Exercise 3

const omitted from parameters for space

Write a **`Circle`** class with three **`float`** fields: the **x** and **y** coordinates of a **`Circle`**'s center, and its **`radius`**.
Fill in the constructors using initialization lists, then implement the **`get`** and **`set`** functions.

Circle.cc

```
#include "Circle.h"

void Circle::SetCenter(float x, float y) {


}

void Circle::ScaleCircle(float scale) {


}
```

Circle.h

```
class Circle {
 public:
  // Default constructor
  // creates a circle of radius 1 at (0,0)
  _____ { }
  // Parameterized constructor
  // Takes in the x, y coords of the center and a radius
  _____ { }
  // Copy constructor
  // Takes in a reference to another circle
  _____ { }
  // Getters
  float get_x() const { _____ }
  float get_y() const { _____ }
  float get_radius() const { _____ }

  // Setters
  // Sets the center of the circle to the given x and y coords
  void SetCenter(float x, float y);
  // Scales the radius of the circle by the given scale factor
  // For simplicity, negative scale factors are allowed.
  void ScaleCircle(float scale);

 private:
  // the location of the center of a circle and its radius
  float x_, y_, rad_;
};  // class Circle
```

Exercise 3

const omitted from parameters for space

Write a **`Circle`** class with three **`float`** fields: the **x** and **y** coordinates of a **`Circle`**'s center, and its **`radius`**.
Fill in the constructors using initialization lists, then implement the **`get`** and **`set`** functions.

Circle.cc

```
#include "Circle.h"

void Circle::SetCenter(float x, float y) {


}

void Circle::ScaleCircle(float scale) {


}
```

Circle.h

```
class Circle {
 public:
  // Default constructor
  // creates a circle of radius 1 at (0,0)
  Circle() : x_(0.0), y_(0.0), rad_(1.0) { }
  // Parameterized constructor
  // Takes in the x, y coords of the center and a radius
  Circle(float x, float y, float rad) : x_(x), y_(y), rad_(rad) {}
  // Copy constructor
  // Takes in a reference to another circle
  Circle(Circle& c) : x_(c.x_), y_(c.y_), rad_(c.rad_) { }
  // Getters
  float get_x() const { _____ }
  float get_y() const { _____ }
  float get_radius() const { _____ }

  // Setters
  // Sets the center of the circle to the given x and y coords
  void SetCenter(float x, float y);
  // Scales the radius of the circle by the given scale factor
  // For simplicity, negative scale factors are allowed.
  void ScaleCircle(float scale);

 private:
  // the location of the center of a circle and its radius
  float x_, y_, rad_;
};  // class Circle
```

Exercise 3

const omitted from parameters for space

Write a **`Circle`** class with three **`float`** fields: the **x** and **y** coordinates of a **`Circle`**'s center, and its **`radius`**.
Fill in the constructors using initialization lists, then implement the **`get`** and **`set`** functions.

Circle.cc

```
#include "Circle.h"

void Circle::SetCenter(float x, float y) {


}

void Circle::ScaleCircle(float scale) {


}
```

Circle.h

```
class Circle {
 public:
  // Default constructor
  // creates a circle of radius 1 at (0,0)
  Circle() : x_(0.0), y_(0.0), rad_(1.0) { }
  // Parameterized constructor
  // Takes in the x, y coords of the center and a radius
  Circle(float x, float y, float rad) : x_(x), y_(y), rad_(rad) {}
  // Copy constructor
  // Takes in a reference to another circle
  Circle(Circle& c) : x_(c.x_), y_(c.y_), rad_(c.rad_) { }
  // Getters
  float get_x() const { return x_; }
  float get_y() const { return y_; }
  float get_radius() const { return rad_; }

  // Setters
  // Sets the center of the circle to the given x and y coords
  void SetCenter(float x, float y);
  // Scales the radius of the circle by the given scale factor
  // For simplicity, negative scale factors are allowed.
  void ScaleCircle(float scale);

 private:
  // the location of the center of a circle and its radius
  float x_, y_, rad_;
};  // class Circle
```

Exercise 3

const omitted from parameters for space

Write a **`Circle`** class with three **`float`** fields: the **x** and **y** coordinates of a **`Circle`**'s center, and its **`radius`**.
Fill in the constructors using initialization lists, then implement the **`get`** and **`set`** functions.

Circle.cc

```cpp
#include "Circle.h"

void Circle::SetCenter(float x, float y) {
  x_ = x;
  y_ = y;
}

void Circle::ScaleCircle(float scale) {
  rad_ = rad_ * scale;

}
```

Circle.h

```cpp
class Circle {
 public:
  // Default constructor
  // creates a circle of radius 1 at (0,0)
  Circle() : x_(0.0), y_(0.0), rad_(1.0) { }
  // Parameterized constructor
  // Takes in the x, y coords of the center and a radius
  Circle(float x, float y, float rad) : x_(x), y_(y), rad_(rad) {}
  // Copy constructor
  // Takes in a reference to another circle
  Circle(Circle& c) : x_(c.x_), y_(c.y_), rad_(c.rad_) { }
  // Getters
  float get_x() const { return x_; }
  float get_y() const { return y_; }
  float get_radius() const { return rad_; }

  // Setters
  // Sets the center of the circle to the given x and y coords
  void SetCenter(float x, float y);
  // Scales the radius of the circle by the given scale factor
  // For simplicity, negative scale factors are allowed.
  void ScaleCircle(float scale);

 private:
  // the location of the center of a circle and its radius
  float x_, y_, rad_;
};  // class Circle
```