Please note that some of the resources used in this assignment require a Stanford Network Account and therefore may not be accessible.

Assignment 2: Six Degrees of Kevin Bacon

Craving a little Oscar trivia? Try your hand in an Internet parlor game about Kevin Bacon's acting career. He's never been nominated for an Oscar, but he's certainly achieved immortality—based on the premise that he is the hub of the entertainment universe. Mike Ginelli, Craig Fass and Brian Turtle invented the game while students at Albright College in 1993, and their Bacon bit spread rapidly after convincing then TV talk-show host Jon Stewart to demonstrate the game to all those who tuned in. From these humble beginnings, a Web site was built, a book was published and a nationwide cult-fad was born.

When you think about Hollywood heavyweights, you don't immediately think of Kevin Bacon. But his career spans almost 20 years through films such as *Flatliners*, *The Air Up There*, *Footloose*, *The River Wild*, *JFK* and *Animal House*. So brush up on your Bacon lore. To play an Internet version, visit **`http://oracleofbacon.org/`**.

Due: Thursday, April 17th at 11:59 p.m.

How to Play

The game takes the form of a trivia challenge. Propose two names, and your friend/opponent has to come up with a sequence of movies and mutual co-stars connecting the two. In this case, your opponent takes on the form of your computer, and the computer is exceptionally good.

Jack Nicholson and Meryl Streep? That's easy:

```
Actor or actress [or <enter> to quit]: Jack Nicholson
Another actor or actress [or <enter> to quit]: Meryl Streep

   Jack Nicholson was in "Heartburn" (1986) with Meryl Streep.
```

Mary Tyler Moore and Red Buttons? Hmmm… not so obvious:

```
Actor or actress [or <enter> to quit]: Mary Tyler Moore
Another actor or actress [or <enter> to quit]: Red Buttons

   Mary Tyler Moore was in "Change of Habit" (1969) with Regis Toomey.
   Regis Toomey was in "C.H.O.M.P.S." (1979) with Red Buttons.
```

Barry Manilow and Lou Rawls?

```
Actor or actress [or <enter> to quit]: Barry Manilow
Another actor or actress [or <enter> to quit]: Lou Rawls

   Barry Manilow was in "Bitter Jester" (2003) with Dom Irrera.
   Dom Irrera was in "Man Is Mostly Water, A" (2000) with Lou Rawls.
```

It's the people you've never heard of that are far away from each other:

```
Actor or actress [or <enter> to quit]: Carol Eby
Another actor or actress [or <enter> to quit]: Debra Muubu

   Carol Eby was in "Bottega dell'orefice, La" (1988) with Burt Lancaster.
   Burt Lancaster was in "Scalphunters, The" (1968) with Tony Epper (I).
   Tony Epper (I) was in "Alien from L.A." (1988) with Debra Muubu.
```

Is it true? Your buffoon of a lecturer has a Bacon number of 3?

```
Actor or actress [or <enter> to quit]: Jerry Cain
Another actor or actress [or <enter> to quit]: Kevin Bacon

   Jerry Cain was in "No Rules" (2005) with Dian Bachar.
   Dian Bachar was in "Rocky & Bullwinkle" (2000) with Robert De Niro.
   Robert De Niro was in "Sleepers" (1996) with Kevin Bacon.
```

I have no idea who this particular Jerry Cain is. Maybe you do.

Overview

There are two major components to this assignment:

- You need to provide the implementation for an **`imdb`** class[1], which allows you to quickly look up all of the films an actor or actress has appeared in and all of the people starring in any given film. We *could* layer our **`imdb`** class over two STL **`map`**s—one mapping people to movies and another mapping movies to people—but that would require we read in several megabytes of data from flat text files. That type of configuration takes several minutes, and it's the opposite of fun if you have to sit that long before you play. Instead, you'll tap your sophisticated understanding of memory and data representation in order to look up movie and actor information very, very quickly. This is the meatier part of the assignment, and I'll get to it in a moment.

- You also need to implement a **breadth-first search algorithm** that consults your super-clever **`imdb`** class to find the shortest path connecting any two actor/actresses. If the search goes on for so long that you can tell it'll be of length 7 or more, then you can be reasonably confident (and pretend that you know for sure that) there's no path connecting them. This part of the assignment is more CS106B-like, and it's a chance to get a little more experience with the STL and to see a legitimate scenario where a complex program benefits from two types paradigms: high-level C++ (with its templates and its object orientation) and low-level C (with its exposed memory and its procedural orientation.)

---

[1] **`imdb`** is short for Internet Movie Database; our name is a gesture to the company that provides all of the data for the hundreds of thousands of movies and movie stars.

Task I: The `imdb` class

First off, I want to you complete the implementation of the **`imdb`** class. Here's the interface:

```
struct film {
   string title;
   int year;
};

class imdb {
   public:
      imdb(const string& directory);
      bool getCredits(const string& player, vector<film>& films) const;
      bool getCast(const film& movie, vector<string>& players) const;
     ~imdb();

   private:
      const void *actorFile;
      const void *movieFile;
};
```

The constructor and destructor have already been implemented for you, because the manner in which I initialize the **`actorFile`** and **`movieFile`** fields to address the raw data representations uses some nontrivial UNIX. They each take O(1) time to run, because typically you want constructors and destructors to be as lightweight as possible. You need to implement the **`getCredits`** and **`getCast`** methods by manually crawling over these raw data representations in order to produce vectors of films and actor names. When properly implemented, they provide lightning-speed access to a gargantuan amount of information.

Understand up front that you are implementing these two methods to crawl over two arrays of bytes in order to synthesize data structures for the client. What follows below is a description of how the memory is laid out. You aren't responsible for creating the data files in any way; you're just responsible for understanding how everything is encoded so that you can re-hydrate information from byte-level representations.

The Raw Data Files

The **`actorFile`** and **`movieFile`** fields each address gigantic blocks of memory. They are each configured to point to mutually referent databases, and the format of each is described below. The **`imdb`** file constructor sets these pointers up for you, so you can proceed as if everything is set up for **`getCast`**/**`Credits`** to just run.

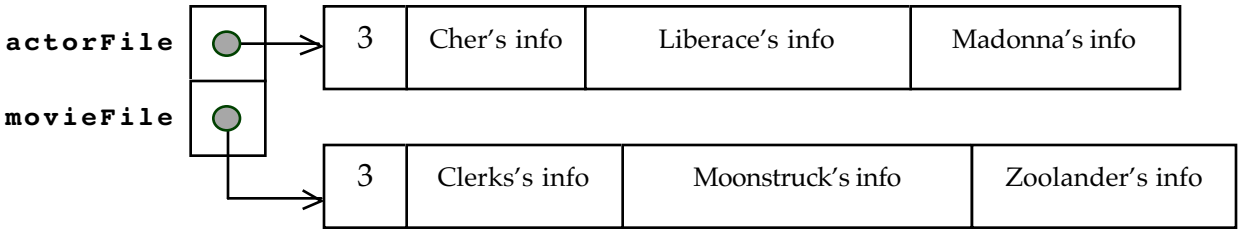
For the purposes of illustration, let's assume that Hollywood has produced a mere three movies, and that they've always rotated through the same three actors whenever the time came to cast their three films. Let's pretend those three films are as follows:

> Clerks, released in 1993, starring Cher and Liberace.
> Moonstruck, released in 1988, starring Cher, Liberace, and Madonna.

Zoolander, released in 1999, starring Liberace and Madonna.

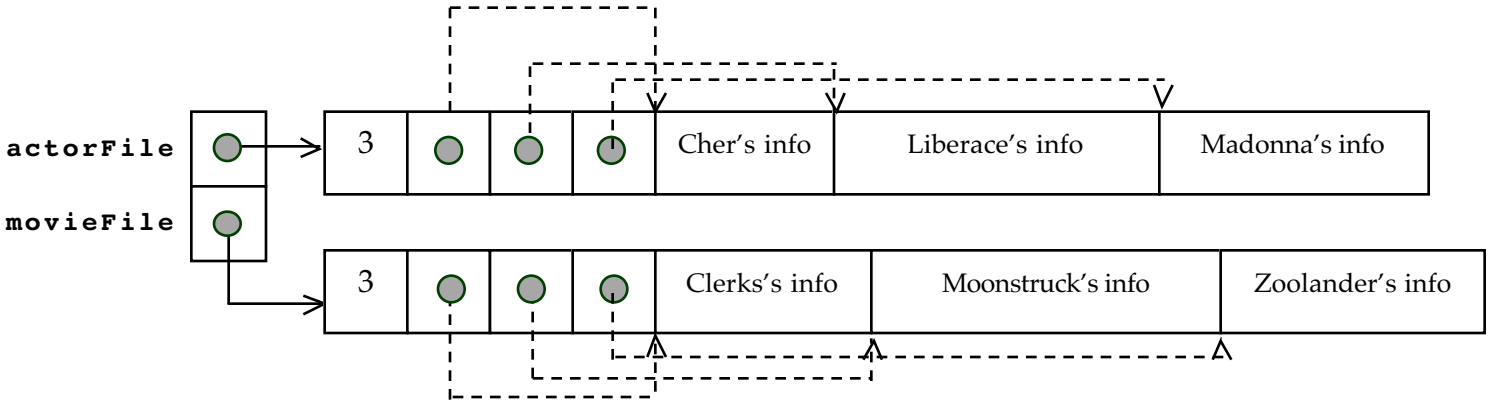
Remember, we're pretending.

If the **`imdb`** were configured to store the above information, you could imagine its **`actorFile`** and **`movieFile`** fields being initialized (by the constructor I already wrote for you) as follows:



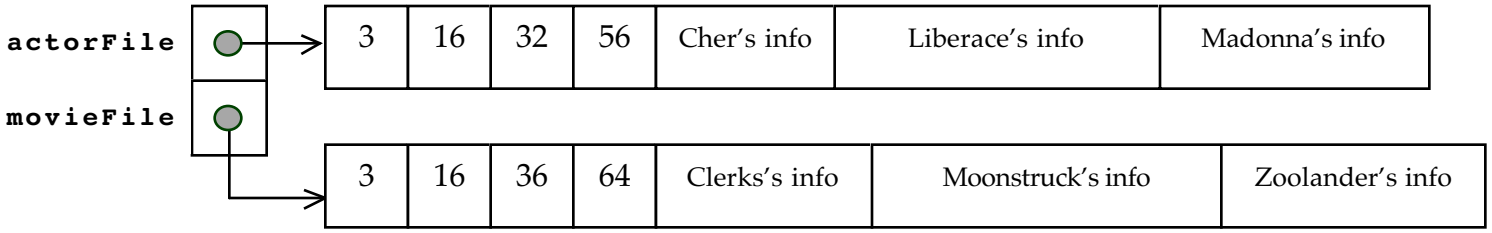
However, each of the records for the actors and the movies will be of variable size. Some movie titles are longer than others; some films feature 75 actors, while others star only a handful. Some people have prolific careers, while several people are one-hit wonders. Defining a **`struct`** or **`class`** to overlay the blocks of data would be a fine idea, except that doing so would constrain all records to be the same size. We don't want that, because we'd be wasting a good chunk of memory when storing information on actors who appeared in just one or two films (and for films that feature just a handful of actors.)

However, by allowing the individual records to be of variable size, we lose our ability to binary search a sorted array of records. The number of actors is well over 300,000; the number of movies is some 124,000, so linear search would be **slow**. All of the actors and movies are sorted by name (and then by year if two movies have the same name), so binary search is still within our reach. The strong desire to search quickly motivated my decision to format the data files like this:



Spliced in between the number of records and the records themselves is an array of integer offsets. They're drawn as pointers, but they really aren't stored as pointers. We want the data images to be relocatable—that is, we want the information stored in the

data images pointed to by **actorFile** and **movieFile** to be useful, regardless of what addresses get stored there. By storing integer offsets, we can manually compute the location of Cher's record, Madonna's record, or Clerk's record, etc, by adding the corresponding offsets to whatever **actorFile** or **movieFile** happens to be. A more accurate picture of what gets stored (and this is really what the file format is) is presented here.



Because the numbers are what they are, we would expect Cher's 16-byte record to sit 16 bytes from the front of **actorFile**, Liberace's 24-byte record to sit 32 bytes within the **actorFile** image, and so forth. Looking for Moonstruck? Its 28-byte record can be found 36 bytes ahead of whatever address is stored in **movieFile**. Note that the actual offsets tell me where records are relative to the base address, and the **deltas** between offsets tell me how large the actual records are.

Because all of the offsets are stored as four byte integers, and because they are in a sense sorted if the records they reference are sorted, we can use binary search. Woo!

To summarize:

- **actorFile** points to a large mass of memory packing all of the information about all of the actors into one big blob. The first four bytes store the number of actors (as an **int**); the next four bytes store the offset to the zeroth actor, the next four bytes store the offset to the first actor, and so forth. The last offset is followed by the zeroth record, then the first record, and so forth. The records, even though variable in length, are sorted by name.
- **movieFile** also points to a large mass of memory, but this one packs the information about all films ever made. The first four bytes store the number of movies (again, as an **int**); the next ***(int *)movieFile * 4** bytes store all of the **int** offsets, and then everything beyond the offsets is real movie data. The movies are sorted by title, and those sharing the same title are sorted by year.
- The above description above generalizes to files with 300,000 actors and 100,000 movies. The rules are the same.

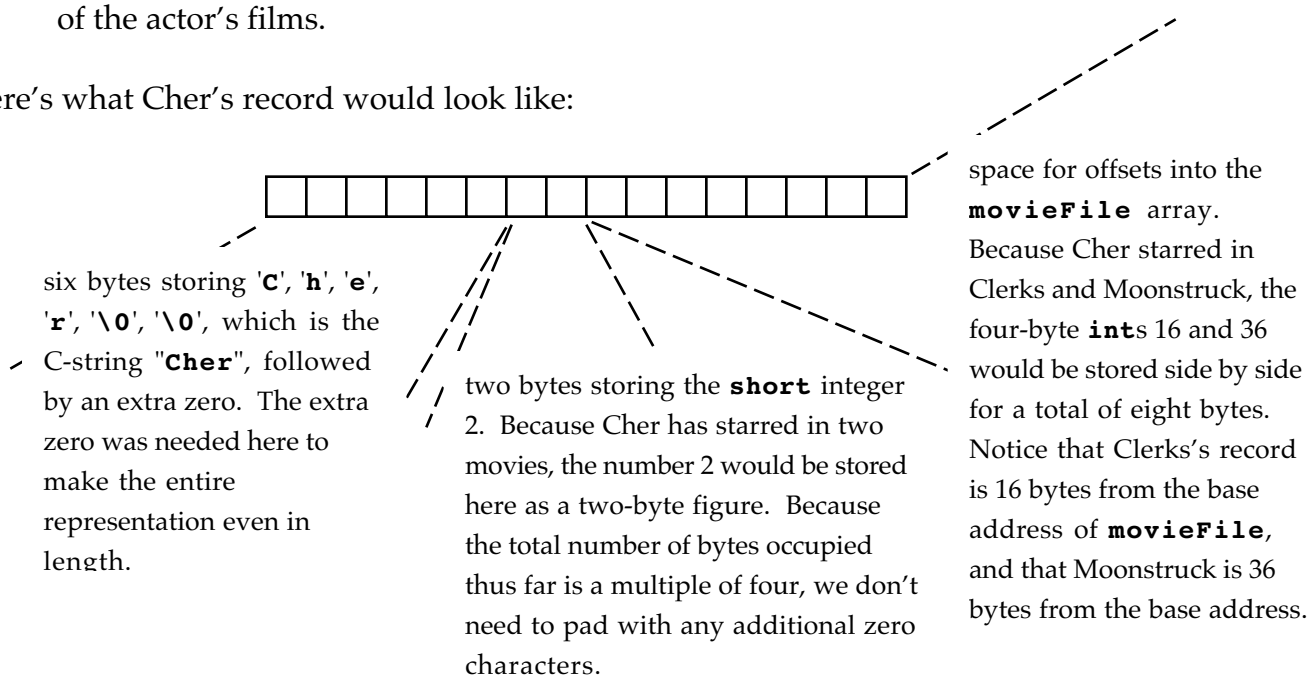
The Actor Record

The actor record is a packed set of bytes collecting information about an actor and the movies he's appeared in. We don't use a **struct** or a **class** to overlay the memory associated with an actor, because doing so would constrain the record size to be

constant for all actors. Instead, we lay out the relevant information in a series of bytes, the number of which depends on the length of the actor's name and the number of films he's appeared in. Here's what gets manually placed within each entry:

1. The name of the actor is laid out character by character, as a normal null-terminated C-string. If the length of the actor's name is even, then the string is padded with an extra '\0' so that the total number of bytes dedicated to the name is always an even number. The information that follows the name is most easily interpreted as a short integer, and virtually all hardware constrains any address manipulated as a **`short *`** to be even.
2. The number of movies in which the actor has appeared, expressed as a two-byte short. (Some people have been in more than 255 movies, so a single byte just isn't enough.) If the number of bytes dedicated to the actor's name (always even) and the short (always 2) isn't a multiple of four, then two additional '\0''s appear after the two bytes storing the number of movies. This padding is conditionally done so that the 4-byte integers than follow sit at addresses that are multiples of four.
3. An array of offsets into the **`movieFile`** image, where each offset identifies one of the actor's films.

Here's what Cher's record would look like:



The Movie Record

The movie record is only slightly more complicated. The information that needs to be compressed is as follows:

1. The title of the movie, terminated by a **`'\0'`** so the character array behaves as a normal C-string.

2. The year the film was released, expressed as a single byte. This byte stores the year – 1900. Since Hollywood is less than 2^8 years old, it was fine to just store the year as a delta from 1900. If the total number of bytes used to encode the name and year of the movie is odd, then an extra '\0' sits in between the one-byte year and the data that follows.
3. A two-byte **short** storing the number of actors appearing in the film, padded with two additional bytes of zeroes if needed.
4. An array of four-byte integer offsets, where each integer offset identifies one of the actors in the **actorFile**. The number of offsets here is, of course, equal to the short integer read during step 3.

One major gotcha: Some movies share the same title even though they are different. (The Manchurian Candidate, for instance, was first released in 1962, and then remade in 2004. They're two different films with two different casts.) If you look in the **imdb-utils.h** file, you'll see that the **film** struct provides **operator<** and **operator==** methods. That means that two **film**s know how to compare themselves to each other using infix **==** and **<** (though not using **!=**, **>**, **>=**, or **<=**). You can just rely on the **<** and **==** to compare two **film** records. In fact, you **should**, because the movies in the **movieData** binary image are sorted to respect **film::operator<**.

It's best to work on the implementation of the **imdb** class in isolation, not worrying about the details of the search algorithm you'll eventually need to write. I've provided a test harness to exercise the **imdb** all by itself, and that code sits in **imdb-test.cc**. The make system generates an test application called **imdb-test** which you can use to verify that your **imdb** implementation is solid. I provide sample versions of this **imdb-test** thing for both Solaris and for Linux, so you can run your version and my version side by side and make sure they match character for character.

Task II: Implementing Search

You're back in C++ mode. At this point, I'm assuming your **imdb** class just works, and the fact that there's some exceedingly shrewd pointer gymnastics going on in the **imdb.cc** file is completely disguised by the simple **imdb** interface. Use the services of your **imdb** and my **path** to implement a breadth-first search for the shortest possible path. Leverage off the STL containers as much as possible to get this done. Here are the STL classes I used in my solution:

- **vector<T>**: there's no escaping this one, because the **imdb** requires we pull films and actors out of the binary images as **vector**s.
- **list<T>**: The **list** is a doubly-linked list that provides O(1) **push_back**, **front**, and **pop_front** operations. There's also a **queue** template, and you can use that if you want, but I'm so bugged that the STL **queue** calls its methods **push** and **pop** instead of **enqueue** and **dequeue** that I boycotted it and used the **list** instead.

- **set<T>**: I used two **set**s to keep track of previously used actors and films. If you're implementing a breadth-first search and you encounter a movie or actor that you've seen before, there's no reason to use it/him/her a second time. You shouldn't need to use anything other than **set<T>::insert**.

The **dinkumware** web site provide a clear, nicely formatted presentation of the list, vector, and set templates. You don't need to read up on every method—just the ones you know have to exist in order for them to be useful.

Here's the general algorithm I used for my own **generateShortestPath** function:

```
list<path> partialPaths; // functions as a queue
set<string> previouslySeenActors;
set<film> previouslySeenFilms;

create a partial path around the start actor;
add this partial path to the queue of partial paths;
while queue isn't empty and its front element is of length 5 or less
   pull off front path (involves both front and pop_front)
   look up last actor's movies
   for each movie in his/her list of movies you've not seen before
      add movie to the set of previously seen movies
      look up movie's cast
      for each cast member you've not seen before
         add cast member to set of those previously seen
         clone the partial path
         add the movie/costar connection to the clone
         if you notice the costar is your target, then print the path and return
         otherwise add this new partial path to the end of the queue

if the while loop ends, print that you didn't find a path
```

There are many clever optimizations that can be made, and I go through a few of them below. But you're only expected to implement a search that's consistent with the algorithm above.

How To Proceed

Seven days is a good stretch of time, but you have a lot to do. If you aren't proactive in making the development process as easy as possible, you're going to end up spending twice as much time as everyone else. Here's the best advice I can give you:

- Compile the starting files to see how they work. Read all of the provided interface files to understand not only what they do, but also how they will contribute to the final product. Work on the **imdb** class first, test it using the **imdb-test.cc** file, and work to match my sample application's output exactly. Only after you've nailed the **imdb** implementation should be move on to the search.

- Development incrementally. Divide the process up to include several intermediate milestones (and when I say several, I mean on the order of 20 or 30.) Let your code base evolve into the program it needs to be.
- Compile and test often. Never write more than a few new lines of code without compiling and testing to see that your changes work as intended. In general, I never write more than 10 – 15 lines without compiling and executing to see how my code changed. If you write 100 lines of code, I guarantee you'll have more compiler errors than you can count on both hands. You don't want that. You just don't.

Getting started

Create a local working directory in your leland space where you'd like to consolidate all of your Assignment 2 files. Then type the following commands:

```
> cp -r /usr/class/cs107/assignments/assn-2-six-degrees/ .  (note the dot)
> cd assn-2-six-degrees
```

All of the Assignment 2 starter files will be copied into the current directory. In particular, you will see a **`Makefile`**, several header files, and several source files—all of which will contribute to your program development efforts. Here's a list of the files that pertain to each task:

Task I

Here's the subset of all the files that pertain to just the first of the two tasks:

`imdb-utils.h`	The definition of the **`film`** struct, and an inlined function that finds the data files for you. You shouldn't need to change this file.
`imdb.h`	The interface for the **`imdb`** class. You shouldn't change the public interface of this file, though you're free to change the private section if it makes sense to.
`imdb.cc`	The implementation of the **`imdb`** class constructor, destructor, and methods. This is where your code for **`getCast`** and **`getCredits`** belongs.
`imdb-test.cc`	The unit test code we've provided to help you exercise your **`imdb`**. You shouldn't have to change this file. We've provided sample applications called **`imdb-test-solaris`** and **`imdb-test-linux`** so you know what you're working toward.
`Makefile`	By typing **`make imdb-test`**, you'll compile just the files needed to build **`imdb-test`**. You shouldn't need to change the **`Makefile`** at all.

Task II

Everything from Task I (except **imdb-test.cc**) contributes to the overall **six-degrees** application. Type **make six-degrees** to build the **six-degrees** executable without building the **imdb-test** application (or you can just type **make** and build both.) There are sample **six-degrees-solaris** and **six-degrees-linux** applications for you to play with. In addition to the files used for Task I, there are these:

six-degrees.cc	The file where most if not all of your Task II changes should be made.
path.h	The definition of the **path** class, which is a custom class useful for building paths between two actors. You're free to add methods if you think it's sensible to do so.
path.cc	The implementation of the **path** class. Again, you can add stuff here if you think it makes sense to.

Speeding It Up

You're more than encouraged (though not required) to optimize the search to find paths—particularly those between actors who are far apart from one another—much more quickly. Consider how neat it'd be if the application:

- starts the search from the actor with the smaller number of movies, and reversing the final path if need be. You'll note that the path class provides a nice little **reverse** method for just this. In fact, your sample application does this (though you don't need to if you don't want to.)
- caches the results of all **imdb** method calls using local **map**s—one of type **map<string, vector<film> >**, and a second of type (can you guess?) **map<film, vector<string> >**. Before going to the **imdb**, you can see whether or not you've looked up the same actor or movie on previous searches. If you have, you can more quickly return the stored result without committing to the more intense **getCredits** or **getCast** call.
- uses a bidirectional search from both actor endpoints, building larger and larger search trees from each of them, and seeing if any of their branches touch. This is a very effective way of focusing the search so that paths of length 4, 5, and 6 can be more quickly discovered. Email me if you want some more details on this one. This last optimization is substantial and can get you some extra credit if done nicely.

Electronic submission

You'll submit this assignment and all subsequent programming assignments they way you submitted RSG. Just type **/usr/class/cs107/bin/submit** and you can't go wrong.