Deliverable 1: Six Degrees of Kevin Bacon

Due: Saturday, Sunday 14th at 23:59:59

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 (US) 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 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://www.oracleofbacon.org.

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.
```

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.
```

Sometimes names have a Roman numeral after them, as per the information in the database. Would you get this Australian connection?

```
   Actor or actress [or <enter> to quit]: Kate Ritchie (I)
   Another actor or actress [or <enter> to quit]: Emma Watson (II)

      Kate Ritchie (I) was in "Mere Oblivion" (2007) with Burleigh Smith.
      Burleigh Smith was in "Two Fists, One Heart" (2008) with Rosemary Lenzo.
      Rosemary Lenzo was in "Crush" (2001) with Derek Deadman.
      Derek Deadman was in "Harry Potter and the Sorcerer's Stone" (2001) with
Emma Watson (II).
```

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 we'll get to it in a moment.

  This part will give you a chance to refresh your knowledge of the low-level subset of features available C++ (i.e., the C features). One of the strengths of C++ is that it can operate at low-level if performance is paramount.

- You also need to implement a **breadth-first search algorithm (BFS)** that consults your super-clever **`imdb`** class to find the shortest path connecting any two actors/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 will give you a chance to get experience working with the STL and to see a legitimate scenario where a complex program benefits from two programming paradigms: high-level C++ (with its templates and its object orientation) and low-level C++ (with its exposed memory and its procedural approach).

[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.

Part I: The `imdb` Class

First off, we 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 we 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 **`vector`**s 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.

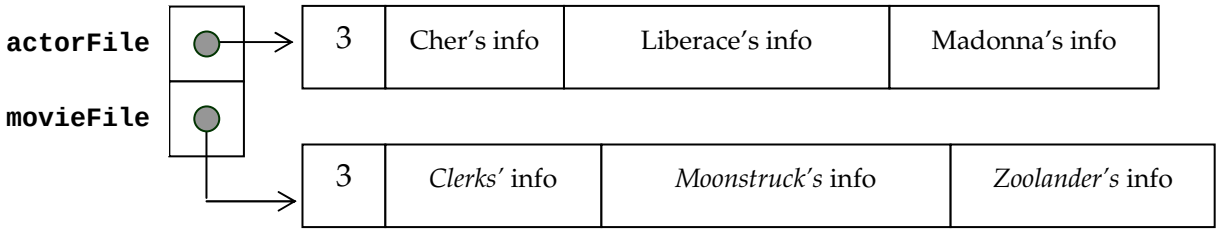
For the purposes of illustration, let's assume that Hollywood has produced a mere three movies, and that they've always rotated through the same three actors whenever the time came to cast their three films.

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.

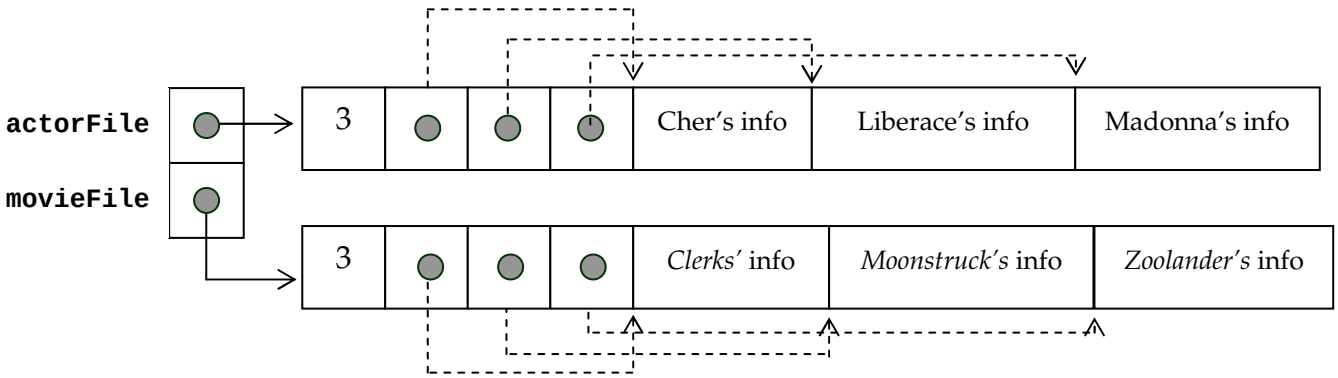
Remember, we're pretending.

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

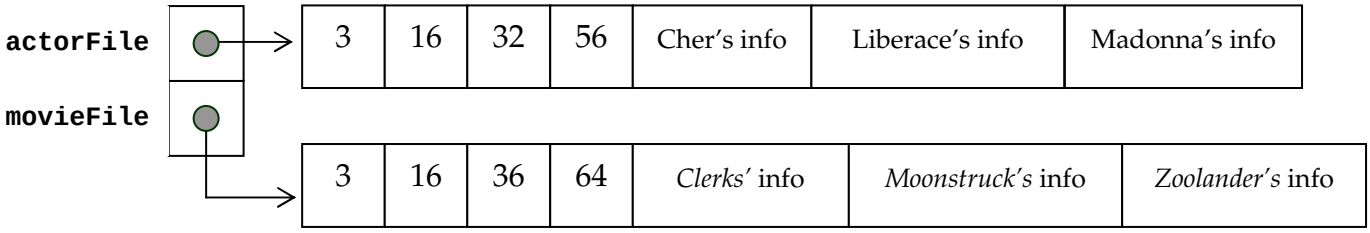


However, each of the records for the actors and the movies will be of variable size. Some movie titles are longer than others; some films feature 75 actors, while others star only a handful. Some people have prolific careers, while several people are one-hit wonders. Defining a **`struct`** or **`class`** 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 over 1,000,000; the number of movies is over 300,000, 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 *Clerks*' record, etc., by adding the corresponding offsets to whatever **`actorFile`** or **`movieFile`** happens to be. A more accurate picture of what gets stored (this is really the file format) 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** (i.e., differences) 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:

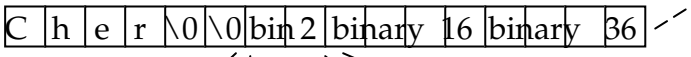
- **`actorFile`** points to a large mass of memory packing all of the information about all of the actors into one big blob. The first four bytes store the number of actors (as an **`int`**); the next four bytes store the offset to the first actor, the next four bytes store the offset to the second actor, and so forth. The last offset is followed by the first record, then the second 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`**); then come 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 1,000,000 actors and 300,000 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. Aligned storage is required by many architectures.
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:



six bytes storing '**`C`**', '**`h`**', '**`e`**', '**`r`**', '**`\0`**', '**`\0`**', which is the C-string "**`Cher`**", followed by an extra zero. The extra zero was needed here to make the entire representation even in length.

two bytes storing the **`short`** integer 2. Because Cher has starred in two movies, the number 2 would be stored here as a two-byte figure. Because the total number of bytes occupied thus far is a multiple of four, we don't need to pad with any additional zero characters.

space for offsets into the **`movieFile`** array. Because Cher starred in *Clerks* and *Moonstruck*, the four-byte **`int`**s 16 and 36 would be stored side by side for a total of eight bytes. Notice that *Clerks'* record is 16 bytes from the base address of **`movieFile`**, and that *Moonstruck* is 36 bytes from the base address.

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. We've provided a test harness to exercise the **`imdb`** all by itself, and that code sits in **`imdb-test.cpp`**. The make system generates an test application called **`imdb-test`** which you can use to verify that your **`imdb`** implementation is solid.

We will provide a sample version of this **`imdb-test`** thing (which you will only be able to run from the subject directory), so you can run your version and our version side by side and make sure they match character for character.

Part II: Implementing Search

You're back in C++ mode. At this point, we're assuming your **imdb** class just works, and the fact that there's some exceedingly shrewd pointer gymnastics going on in the **imdb.cpp** file is completely disguised by the simple **imdb** interface. Use the services of your **imdb** class and the supplied **path** class to implement a breadth-first search for the shortest possible path. This is a standard algorithm. Here is an outline:

1. Enqueue the root node.
2. Dequeue a node and examine it.
    - If the element sought is found in this node, quit the search and return a result.
    - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node has been examined – quit the search.
4. Repeat from Step 2.

A correct implementation should produce the same results as the sample solution. Leverage off the STL containers as much as possible to get this done. Here are the STL classes used in the sample 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.
- **queue<T>**: the queue data structure naturally lends itself to the BFS algorithm; C++ provides it as the **queue** container adaptor.
- **set<T>**: two **set**s are used to keep track of previously seen actors and films.

The textbook and the C/C++/STL Ref. link on the course website provide suitable references for container usage and available operations.

Important: As has been hinted already and as suggested by the title, if you do not find a connection after searching six levels deep then you should assume that there is no connection and abort the search!

How To Proceed

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 we 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.cpp** 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.

- Develop incrementally. Divide the process up to include several intermediate milestones (and when we say several, we 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, never write more than 10 – 15 lines without compiling and executing to see how the code has changed. If you write 100 lines of code, guaranteed you'll have more compiler errors than you can count on both hands. You don't want that!

Getting started

You are provided with a stub that contains the files you need to get started on this deliverable. The stub is available on the subject account. Login to CSE and then type something like this:

```
% mkdir -p cs3171/del1
% cd cs3171/del1
% cp ~cs3171/soft/del1/del1.zip .
% unzip del1.zip
% ls
% make
```

All of the stub files will be uncompressed into the current directory. In particular, you will see a **`Makefile`**, some header files, and some source files—all of which will aid your program development efforts. Here's a list of the files that pertain to each part:

Part I

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

`imdb-utils.h`	The definition of the **`film`** struct. 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.cpp`	The implementation of the **`imdb`** class constructor, destructor, and methods. This is where your code for **`getCast`** and **`getCredits`** belongs.
`imdb-test.cpp`	The unit test code we've provided to help you exercise your **`imdb`**. You shouldn't have to change this file.
`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. Though have a look inside and you may discover some debugging options.

Part II

Everything from Part I (except **imdb-test.cpp**) 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). In addition to the files used for Part I, there are these:

six-degrees.cpp	The file where most if not all of your Part 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.cpp	The implementation of the **path** class. Again, you can add stuff here if you think it makes sense to.

The Data Files

The data files are quite large (~58MB), so you can use them directly from the subject account: **~cs3171/soft/del1/data**
Or you may choose to copy them locally if you can afford the disk quota:

```
% cd ~/cs3171/del1/
% mkdir -p data
% cp ~cs3171/soft/del1/data/* ./data/
```

Run the programs by specifying the path to the data files as argument, for example:

```
% ./imdb-test ~cs3171/soft/del1/data
% ./six-degrees ./data
```

Testing

You are responsible for making sure your implementation is 110% correct, and if some bug in your code isn't flagged by your tests, that's your crisis and not ours. You should try building your own test suite to make sure that everything checks out okay.

We will supply a sample solution that you can use for testing. Your program must behave exactly like the sample solution, since the sample solution output will be used for automarking. The supplied output operator of the **path** class should help with the formatting of your output. Use the **diff** utility to ensure output matches, eyeball won't suffice.

You are free to use any development environment you like, however it is your job to ensure that your submission compiles and runs correctly on the school machines. In particular we will compile your submission (via the **Makefile** provided) with:

```
g++-4.2 -Wall -Werror -O2 ...
```

Marking

This deliverable is worth **10%** of your final mark.

Your submission will be given a mark out of 100 with a 80/100 automarked component for output correctness and a 20/100 manually marked component for code style and quality.

As this is a third-year course we expect that your code will be well formatted, documented and structured. We also expect that you will use standard formatting and naming conventions. However, the style marks will be awarded for writing C++ code rather than C code. Your tutors will check that you have made an effort to find and use the appropriate C++ features/library components in your code.

Note that the supplied code was not written by us and may not display ideal formatting style and/or ideal C++ design and implementation. You are not required to follow the same style, if you feel you can improve upon it.

Submission

Due: Sunday, August 14th at 23:59:59

Copy your code to your CSE account and make sure it compiles without any errors or warnings. Then run your test cases. If all is well then submit using the command (from within your **`del1`** directory):

```
% give cs3171 del1 imdb.[h|cpp] imdb-utils.h six-degrees.cpp path.[h|cpp]
```

Note that you are to submit specific files only, this means that these are the only files you should modify. You are free to introduce any new classes you like. *To make marking easier please place any new classes in the specified submission files.* We will use our original supplied **`Makefile`** to compile your submission.

If you submit and later decide to submit an even better version, go ahead and submit a second (or third, or seventh) time; we'll only mark the last one. Be sure to give yourself more than enough time before the deadline to submit.

Late submissions will be penalised unless you have legitimate reasons to convince the LIC otherwise. Any submission after the due date will attract a reduction of 10% per day to the maximum mark. A day is defined as a 24-hour day and includes weekends and holidays. Precisely, a submission x hours after the due date is considered to be $ceil(x/24)$ days late. No submissions will be accepted more than five days late.

Plagiarism constitutes serious academic misconduct and will not be tolerated. CSE implements its own plagiarism addendum to the UNSW plagiarism policy. You can find it here:

http://www.cse.unsw.edu.au/~chak/plagiarism/plagiarism-guide.html

Further details about lateness and plagiarism can be found in the Course Outline.

Acknowledgements

This assignment, including supplied code, was designed/written by Jerry Cain, Stanford University. We have modified it slightly to match our requirements.

The information used in this assignment is courtesy of the **The Internet Movie Database (IMDb)**, http://www.imdb.com.

Disclaimer

The data files provided are a direct and complete compilation of actor/movie information supplied by IMDb and current in July 2009. We take no responsibility regarding their content and accuracy.

We couldn't keep these data files up to date as rebuilding them would require at least 12GB memory and the resulting data files would be much bigger.

The provided data files are sufficient for the purposes of this assignment.