Assignment 3

Due on Sunday March 11, 11:59pm

This year's OSCAR night is coming. Let's do some work about movies ☺

There is an Internet game about Kevin Bacon's acting career (visit http://oracleofbacon.org/). The game takes the form of a trivia challenge. Ie. 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. Behind the scene, there is an imdb (Internet Movie Database) consisting of 2 important compressed data files containing all actors/actress and movie information. In this assignment, you are going to access and retrieve data from this imdb.

To do so, you need to provide the implementation for an imdb class, which allows you to look up all of the films an actor or actress has appeared in and all of the people starring in any given film. Part of the code is provided as the starter. Here is the interface:

```
struct film {
      string title;
      int year;
};
class imdb {
   public:
      imdb(const string& directory);
      bool getPlayer(const size_t player_idx, vector<film>& films) const;
      bool getFilm(const size_t movie_idx, vector<string>& players) const;
      ~imdb();
   private:
      const void *actorFile;
      const void *movieFile;
};
```

The constructor and destructor have already been implemented to initialize the ***actorFile*** and ***movieFile*** fields to address the raw data representations. You need to implement the ***getPlayer*** and ***getFilm*** methods by manually crawling over these raw data representations in order to produce a vector of films for an actors/actress index, and a vector of actors/actresses for a film index:

Actor/actress idx	Film idx
film 1 film 2 … film m	actor/actress 1 actor/actress 2 … actor/actress n

What follows below is a description of how the memory is laid out. You need to understand how information is encoded so that you can re-hydrate information from byte-level representations.

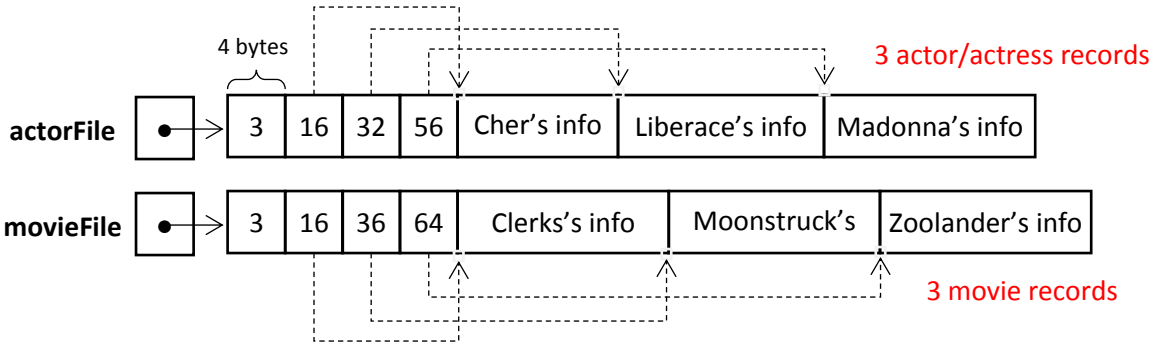
A. The Raw Data Files

The ***actorFile*** and ***movieFile*** fields each address gigantic blocks of memory. They are each configured to point to mutually referent databases. The ***imdb*** constructor has set these pointers up for you, so you can proceed as if everything is set up for ***getPlayer/Film*** to just run.

For the purposes of illustration, let's assume that there are only three movies, and three actors. The 3 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.

Given the nature that some people have prolific careers, while several people are one-hit wonders; some movie titles are longer than others; some films feature many actors, while others star only a handful, defining a fixed record size that fits all would waste memory. Instead, we define each of the records for the actors and the movies with variable size for ***actorFile*** and ***movieFile*** in the following way (being initialized by the constructor for you):



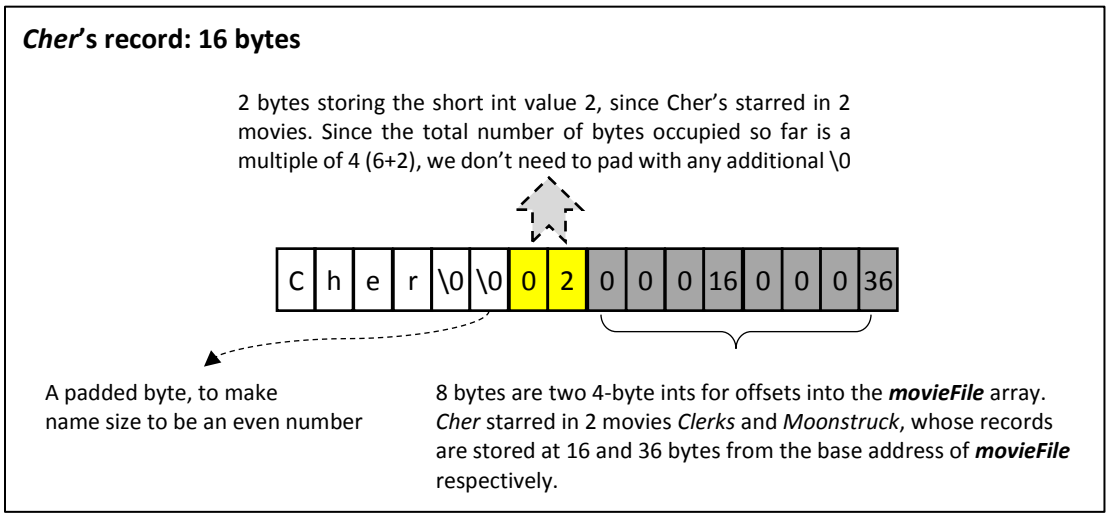
- **actorFile** points to a large mass of memory packing the information about all of the actors into one big blob. The first 4 bytes store the number of actors (as an int); the next 4 bytes store the offset to the zeroth actor, the next 4 bytes store the offset to the first actor, and so forth. The 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.
  For example: *Cher* has 16-byte (32-16) record and sits 16 bytes from the front of ***actorFile***, Liberace has 24-byte (56-32) record and sits 32 bytes within the ***actorFile*** image, and so forth.
- **movieFile** also points to a large mass of memory, but this one packs the information about all films ever made. The first 4 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.
  For example: Moonstruck has 28-byte (64-36) record that 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.

B. The Actor Record (each actor/actress's info)

Each actor record in **actorFile** is a packed set of bytes collecting information about an actor and the movies he/she's appeared in. Similarly, we don't use a fixed length space for each actor record. 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/she's appeared in. Here's what gets manually placed within each entry:

1. **The name of the actor/actress** 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**.
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 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 4, 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 follow sit at addresses that are multiples of 4.
3. An array of offsets into the ***movieFile*** image, where each offset identifies one of the actor's films.

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



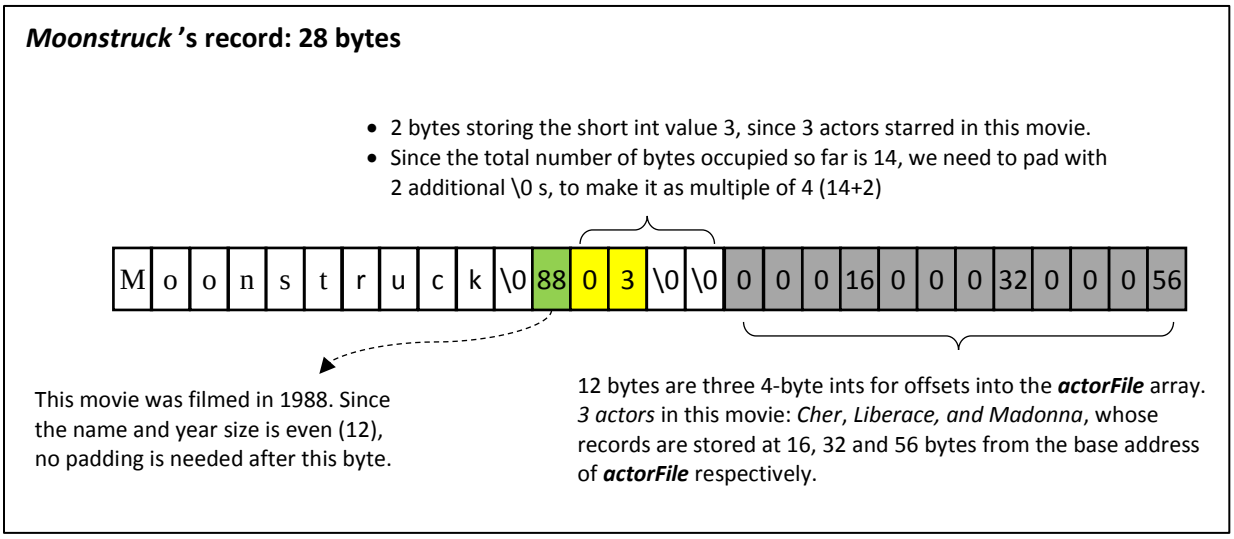
C. The Movie Record (a movie's info)

The movie record is slightly more complicated:

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 2-byte short storing the **number of actors appearing in the film**. Like step 2 in the actor record, If the number of bytes dedicated to the movie name and year (always even) and this short (always 2) isn't a multiple of 4, then two additional '\0's appear after this two bytes storing the number of actors.

4. An array of 4-byte integer offsets, where each integer offset identifies one of the actors in the ***actorFile***. The number of offsets here equals to the short integer read during step 3.

Here's the example for movie *Moonstruck*:



Note: 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 films 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, the movies in the movieData binary image are sorted to respect film::operator<.

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 **getPlayer** and **getFilm** belongs.
- **A3.cc** The main() function that provides simple user interface to receive inputs and calls getPlayer and getFilm.