Assignment 1: Six Degrees of Kevin Bacon

* *This assignment was developed by Jerry Cain*

Craving a little Oscar trivia? Try your hand at an Internet parlor game about Kevin Bacon's acting career. He's never been nominated for an Oscar, but he's still immortal—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 website 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 35 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**.

This assignment is first and foremost a low-level systems programming assignment, but it's also an opportunity to review your C++ while simultaneously exercising your software engineering and low-level memory manipulation skills. You'll also get to see that low-level C coding and high-level C++ data structuring can coexist in the same application.

Due Date: Wednesday, April 10th, 2019 at 11:59pm

No late days can be used on this assignment.

The game takes the form of a trivia challenge: supply any 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 an executable, and that executable is infuriatingly good.

Jack Nicholson and Meryl Streep? That's easy:

```
cgregg@myth60$ ./search "Meryl Streep" "Jack Nicholson (I)"
Meryl Streep was in "Close Up" (2012) with Jack Nicholson (I).
```

Mary Tyler Moore (rest her soul) and Red Buttons? Not so obvious:

```
cgregg@myth60$ ./search "Mary Tyler Moore" "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? Yes!

```
cgregg@myth60$ ./search "Barry Manilow" "Lou Rawls"
Barry Manilow was in "Bitter Jester" (2003) with Dom Irrera.
Dom Irrera was in "A Man Is Mostly Water" (2000) with Lou Rawls.
```

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

```
cgregg@myth60$ ./search "Danzel Muzingo" "Liseli Mutti"
Danzel Muzingo was in "My Day in the Barrel" (1998) with Chala Savino.
Chala Savino was in "Barbershop: The Next Cut" (2016) with Troy Garity.
Troy Garity was in "Sunshine" (2007) with Cliff Curtis (I).
Cliff Curtis (I) was in "Rapa Nui" (1994) with Liseli Mutti.
```

Is it true? Jerry Cain, the lecturer, has a Bacon number of 3?

```
cgregg@myth60$ ./search "Jerry Cain (I)" "Kevin Bacon (I)"
Jerry Cain (I) was in "No Rules" (2005) with Romeo Antonio.
Romeo Antonio was in "Doesn't Texas Ever End" (2009) with Irwin Keyes.
Irwin Keyes was in "Friday the 13th" (1980) with Kevin Bacon (I).
cgregg@myth60$ ./search "Jerry Cain (II)" "Kevin Bacon (I)"
Jerry Cain (II) was in "A Film Is a Film" (2015) with Ron Preston (I).
Ron Preston (I) was in "10 to Midnight" (1983) with Wilford Brimley (I).
Wilford Brimley (I) was in "End of the Line" (1987) with Kevin Bacon (I).
```

I have no idea who these Jerry Cains are, but I promise you Jerry Cain the CS instructor is neither one of them.

Overview

There are two major components to this assignment:

- You need to implement the `imdb` class (`imdb` is short for Internet Movie Database), which helps us discover who starred in what. We **could** layer this `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, even on fast machines, and it's the opposite of fun if you have to sit and watch that long before you play. Instead, you'll tap your sophisticated understanding of data representation and learn about something called memory mapping in order to look up movie and actor information from a prepared data structure that's been saved to disk in its binary form. This is the meatier part of the assignment.

- You need to implement a **breadth-first search algorithm** that enlists your **`imdb`** class to find the shortest path connecting any two actors or 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. In other words -- stop your algorithm at this point. This part of the assignment is more CS106B-like, and it's a chance to get a little more experience with the STL (using **`vector`**s, **`set`**s, and **`list`**s) and to see a legitimate scenario where a complex program benefits from coding in two different paradigms: high-level, object-oriented C++ (with its STL template containers and template algorithms) and low-level, imperative C (with its exposed memory, brought to you by CS107, **`*`**, **`&`**, **`[]`**, and **`->`**).

Task I: The `imdb` class

First off, you should complete the implementation of the **`imdb`** class. Here's the reduced interface:

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

class imdb {
public:
    imdb(const string& directory);
    bool good() const;
    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. All the constructor does is initialize **`actorFile`** and **`movieFile`** fields to point to on-disk data structures using the **mmap**routine you'll learn about later on in the course. Since you haven't used **`mmap`** before, I implemented the constructor and destructor for you.

You'll need to implement the **`getCredits`** and **`getCast`** methods by manually crawling over these binary images in order to produce **`vector`**s of movies and actor names. When properly implemented, they provide lightning-speed access to a gargantuan amount of information, because the information is already compactly formatted in a prepared data structure that permanently lives on the **`myth`** machines.

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 appears below is a description of how that memory is laid out. You aren't responsible for creating the data files in any way, but you are responsible for understanding how everything is encoded so that you can rehydrate information our of its byte-level representation.

The Raw Data Files

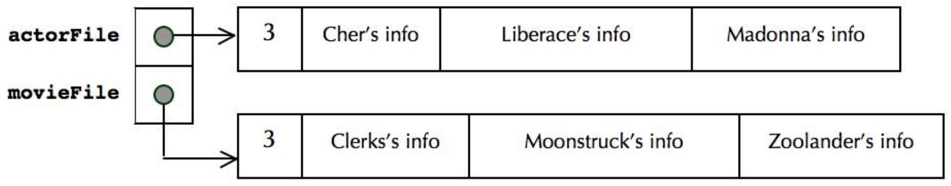
The private **`actorFile`** and **`movieFile`** fields each address large blocks of memory. Each is configured to point to mutually referent database images, and the format of each is described below. The **`imdb`** constructor sets these pointers up for you, so you can proceed as if everything is initialized for **`getCast`** and **`getCredits`** to just work.

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.

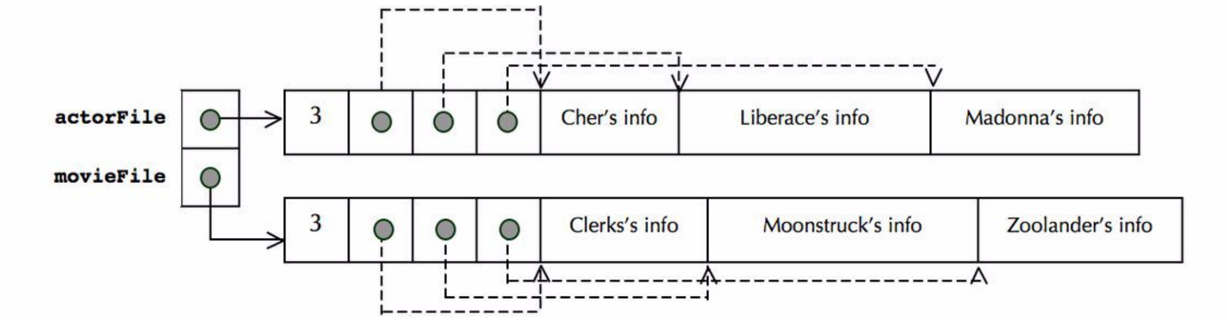
Remember, we're pretending.

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

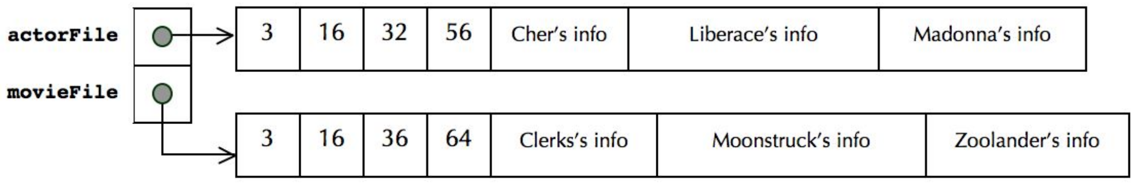


Each of the records for the actors and movies will vary in size. Some movie titles are longer than others; some films feature 75 actors, while others star only one or two. Some actors have prolific careers, while others are one-hit wonders. Defining a **`struct`** or **`class`** to overlay the blocks of data is 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 (hint: via the STL `lower_bound` algorithm) a sorted array of records. The number of actors and actresses is 2.5 million, and the number of movies is just shy of 700,000, so a linear search would be unacceptably slow. All actors and movies are sorted by name (and then by year if two movies have the same name), so binary search is still within reach. The strong desire to binary 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 that way. 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. Restated, we can't embed actual memory addresses in the data images, because the binary image may be loaded into memory at different locations each time an **`imdb`** is created. 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`** turn out 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 offsets tell me where records are relative to the base address, and the **`differences`** between consecutive offsets tell me how large the records are.

Because all of the offsets are stored as **four-byte** integers (and `int`s are four bytes, even on 64-bit systems like the **`myth`**s), and because they are in a sense sorted if the records they reference are sorted, we can use binary search. Rage.

To summarize:

- **`actorFile`** points to a large mass of memory packing all of the information about all of the actors. 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 themselves are sorted by name. I promise.

- **`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*sizeof(int)`** bytes store all of the **`int`** offsets, and everything beyond 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 2,500,000 actors and 700,000 movies.

The Actor Record

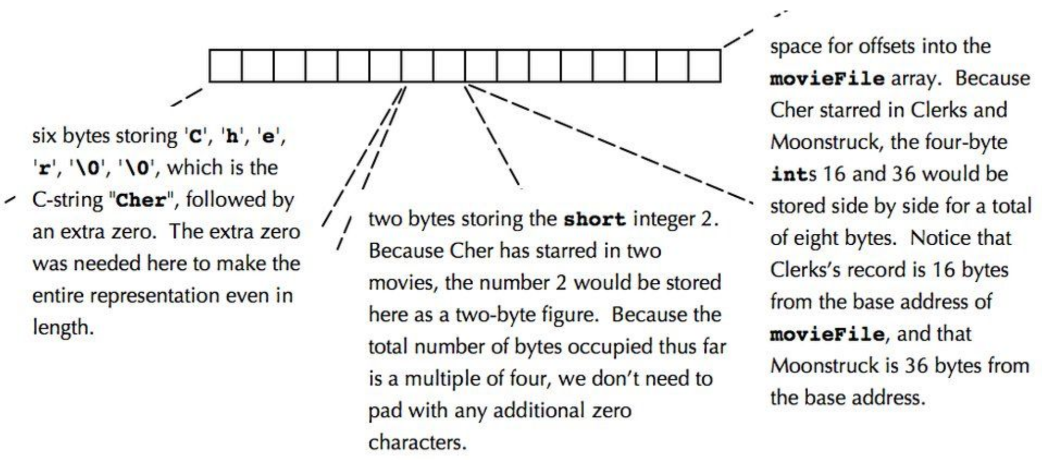
The actor record is a packed set of bytes collecting information about an actor and the movies he or she's appeared in. We don't use a **`struct`** or **`class`** to overlay the memory associated with an actor, because doing so would constrain the record size to be the same for all actors. Instead, we 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 they've 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`**, and the **`myth`**s might constrain addresses manipulated as **`short*`**s 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 isn't always 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 four-byte integers that follow sit at addresses that are multiples of four (again, because the 64-bit **`myth`**'s might be configured to require this).

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 ints 16 and 36 would be stored side by side for a total of eight bytes. Notice that Clerks's 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 is compressed as follows:

1. The **title** of the movie, terminated by a `'\0'`, so the character array behaves as a normal C-string incidentally wedged into a larger binary data figure.

2. The **year** the film was released, expressed as a single byte. This byte stores the year, minus 1900. Since Hollywood is less than 256 years old, it was fine to just store the year as an offset 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 accessible via **`actorFile`**. The number of offsets here is, of course, equal to the **`short`** 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 **`<`**. You can just rely on the **`<`** and **`==`** to compare two **`film`** records. In fact, you **have to**, 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 **`imdbtest.cc`**. The **`make`** system generates a test application called **`imdbtest`** which you can use to verify that your **`imdb`** implementation is solid. I provide my own version in `./samples/imdbtest_soln` (`samples` is a symbolic link in your repo to a shared directory with solution executables) so you can run your version and my version side by side and make sure they match character for character.

Note: Your implementation will be—and in fact is intended to be—an interesting mix of C and C++. You'll be relying on your mad C skills to crawl over these binary images, and you'll be leveraging your C++ mastery to lift that data up into C++ objects. As part of your implementation, you'll need to binary search over the actor and movie offsets to find the actor or movie of interest.

I am **requiring** that you use the STL `lower_bound` algorithm to perform these binary searches, and that you use C++ lambdas (also known as anonymous functions with capture clauses) to provide nameless comparison functions that `lower_bound` can use to guide its search.

Task II: Implementing Search

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

- `vector`: there's no escaping this one, because the `imdb` requires we pull `film`s and `actor`s out of the binary images as `vector`s.
- `list`: 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 and used the `list` instead.
- `set`: 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/them a second time. You shouldn't need to use anything other than `set::insert`.

Getting Code

Go ahead and clone the starter repository that we've set up for you. Do so by typing this:

```
cgregg@myth60$ git clone /usr/class/cs110/repos/assign1/$USER assign1
```

This line places a copy of the starter files dedicated to you in your current working directory. `git` is the name of the source control framework we use to keep track of your changes and minimize the chances you lose any of your work.

Compile often, test incrementally and almost as often as you blink, and run **./tools/submit** when you're done (we will accept the last submission before the deadline, so submit as often as you'd like). As you make progress, you can invoke **./tools/sanitycheck** to commit local changes and run a collection of tests that compare your solution to my own. And be sure your solutions are free of memory leaks and errors, since we'll be running your code through **valgrind**. Note: there's a bug in **valgrind** itself that surfaces when virtually any C++ program is run through it. You can suppress this error by copying the **/usr/class/cs110/tools/config/.valgrindrc** file into your home directory (or copying its contents into an existing **~/.valgrindrc** file):

```
cgregg@myth60$ more /usr/class/cs110/tools/config/.valgrindrc
--memcheck:leak-check=full
--show-reachable=yes
--suppressions=/usr/class/cs110/tools/config/cs110.supp
cgregg@myth60$ cp /usr/class/cs110/tools/config/.valgrindrc ~
```

Assignment 1 Files

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** constructor, destructor, and methods. This is where your code for **getCast** and **getCredits** belongs.

imdbtest.cc

The unit test code we've provided to help you exercise your **imdb**. *You shouldn't have to change this file.*

`Makefile`

By typing **`makeimdbtest,`** you'll compile just the files needed to build **`imdbtest`**. *You shouldn't need to change this file at all.*

Everything from Task I (except **`imdbtest.cc`**) contributes to the overall **`search`** application. Type **`makesearch`** to build the **`search`** executable without building the **`imdbtest`** application (or you can just type **`make`** and build both.) There's a sample executable at **`./samples/search_soln`** for you to play with. Understand that my sample application and yours aren't obligated to publish the same exact shortest path, but you should be sure that the path lengths themselves actually match. In addition to the files used for Task I, there are these:

`search.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.*

Helpful Hints

- Our CS106B and CS106X classes rely on a collection of C++ libraries that aren't generally available outside of Stanford. I want you to code in **standard** C++ instead of the Stanford dialect of it, so you'll need to teach yourself some of the C++ that CS106B/X circumvented. In particular, the CS106 **`Vector`** and **`Map`** templates are officially verboten; you'll instead need to teach yourself the standard **`vector`** and **`map`** classes that ship with all C++ compilers. I've posted a slide deck from prior CS110 offerings that teach some of the C++ I speak of. And you should visit—even bookmark—the landing page of an online C++ documentation site that I think is pretty superb. In particular, pay attention to the documentation for vector, map, list, set, and lower_bound.

- Your implementation of the **`imdb`** class—in particular, its **`getCast`** and **`getCredits`** methods—needs to crawl over memory to resuscitate **`film`**s from the binary data images. Strive to unify common code to helper functions and methods so that you never repeat the same crunchy line of pointer arithmetic more than once. This is particularly important here, because you're very likely to get the pointer arithmetic wrong the first few rounds, and it's important coding errors be confined to one spot instead of being replicated verbatim in several different locations.

- Try to minimize the number of deep copies you make of large data structures (e.g. **`vector`**s, **`map`**s, **`set`**s, etc.) by passing them by reference or by **`const`** reference. In general, I'm happy to pass primitives like **`int`**s and **`bool`**s by value, but anything as large as a **`string`** I generally pass by reference unless I have an exquisite reason for making deep copies.

- Several CS110 graduates have learned the hard way that **`list::size()`** runs in O(n) time instead of O(1) time. That means you don't want to call it in a **`while`** loop to decide whether a list is empty. Instead, rely on the **`list::empty()`** method to do so (it runs in constant time). If you use list::size(), I can guarantee your breadth-first search will run way too slow.

- In past quarters, some students have constructed working solutions without using (or even recognizing the existence of) the supplied **`path`** class. However, your life will be made much easier if you use it, since **`path`**s know how to print themselves out via a custom **`operator<<`**, and it just so happens that **`path`**s print themselves in a way that plays well with our autograder.

- Speaking of the autograder, you should rely on our **`sanitycheck`** tool—again, invoked via **`./tools/sanitycheck`**—to confirm that your solution matches my own for a small suite of test cases. As it turns out, the tests exposed via **`sanitycheck`** are precisely the same tests we use when grading, so if you pass **`sanitycheck`**, then you can expect to pull a 100% on functionality. We won't always expose **`all`** of the tests like we are on Assignment 1, but this time we are.

- If you need a tutorial on how **`valgrind`** works, you can leverage the documentation written for CS107. A full treatise on what **`valgrind`** is and what it can do for you can be found right here. More generic documentation on how to test with **`valgrind`** can be found right here.

- Cool fact time: Mike Krieger, co-founder of Instagram, took CS107 from Jerry about 15 years ago and did this assignment in more or less its current form. Fast forward ten years and Jerry found out that Mike used the same binary data image compression techniques used here to compactly store large amounts of data for the first version of Instagram.