Project 4: BruinNav

Time due: 11 PM, Thursday, March 16

Introduction

The NachenSmall Maps & Navigation Corporation, owner of the popular turn-by-turn navigation website WeHaveTheBestNavigationSoftwareInTheWorldIfYouCanJustRememberTheURL.com, has decided to stop licensing a 3rd-party turn-by-turn navigation system (from a company who's name rhymes with "frugal") and instead build their own navigation engine in-house. Given that the NachenSmall leadership team is comprised entirely of UCLA alums, they've decided to offer the job to build a prototype of the new navigation system the students of CS32. Lucky you!

So, in your last project for CS32, your goal is to build a simple navigation system that loads and indexes a bunch of Open Street Map geospatial data (which contains latitude and longitude data for thousands of streets and attractions) and then build a turn-by-turn navigation system on top of this data. Your completed Project 3 solution should be able to deliver optimal directions like the following, which detail how to get from *1031 Broxton Ave.* (in Westwood) to *The Maltz Park* in (Beverly Hills):

```
You are starting at: 1031 Broxton Avenue
Proceed 0.09 miles southeast on Broxton Avenue
Take a left turn on Kinross Avenue
Proceed 0.07 miles east on Kinross Avenue
Take a right turn on Glendon Avenue
Proceed 0.08 miles southeast on Glendon Avenue
Take a left turn on Lindbrook Drive
Proceed 0.93 miles east on Lindbrook Drive
Take a right turn on Holmby Avenue
Proceed 0.08 miles southeast on Holmby Avenue
Take a left turn on Wilshire Boulevard
Proceed 0.92 miles east on Wilshire Boulevard
Take a left turn on Whittier Drive
Proceed 0.74 miles north on Whittier Drive
You have reached your destination: The Maltz Park
Total travel distance: 2.9 miles
```

If you're able to prove to NachenSmall's reclusive and bizarre CEO, Carey Nachenberg, that you have the programming skills to build a simple navigation engine, he'll hire you to build the full navigation website, and you'll be rich and famous.

Anatomy of a navigation engine

All navigation systems operate on geolocation data, like the data you can find at Open Street Maps project:

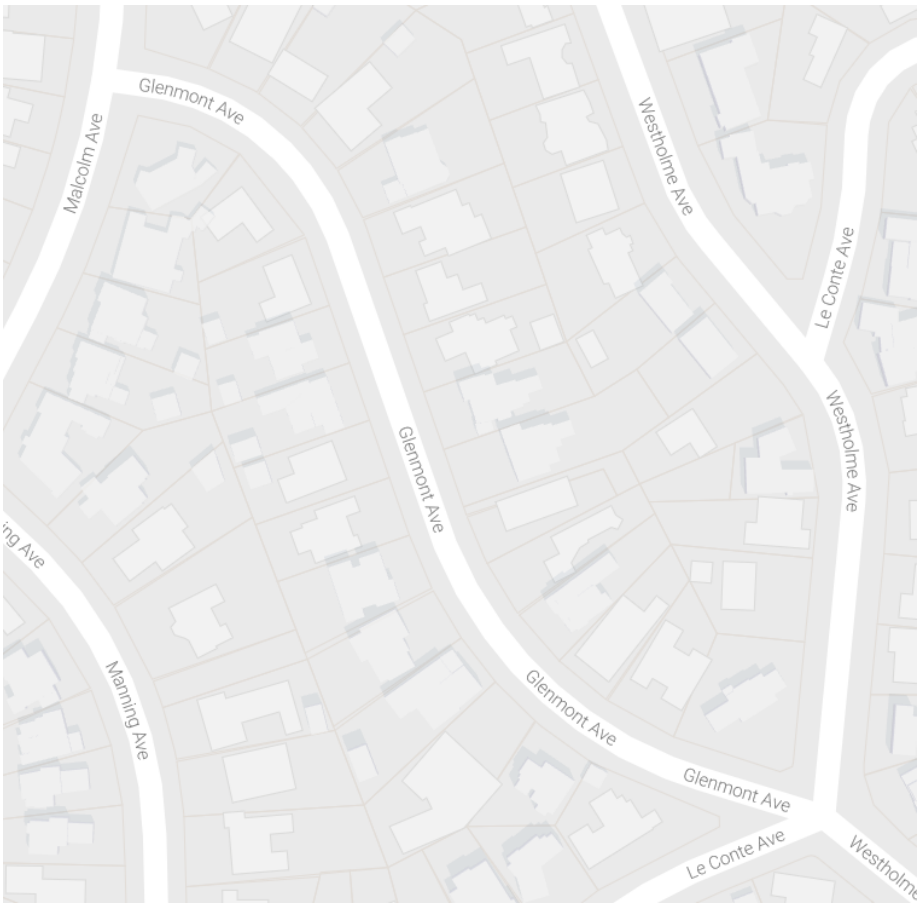
https://www.openstreetmap.org

Open Street Maps (OSM) is an open-source collaborative effort where volunteers can submit street map data to the project (e.g., the geolocations of various streets and attractions), and OSM incorporates this data into its ever-evolving street map database. Companies like Google and TomTom have their own proprietary street map data as well. In this project, we'll be using data from Open Street Maps, because it's freely available.

The OSM data has geolocation (latitude, longitude) data for each street in its map, as well as for attractions (e.g., The Maltz Park, Barney's Beanery, or Engineering VI) in its map. The OSM data also has geolocation data for some street addresses (e.g., 1031 Broxton Ave), although since each address must be added manually by a contributor, the database contains few such street addresses. Each street is broken up into multiple line segments to capture the contours of the street. As you'll see, even a simple street like Glenmont Ave (which is a short street that is just one block long) may be broken up into many segments. This is done to capture the curvy contours of the street, since each individual segment can only represent a straight line. For example, here are the segments from OSM for Glenmont Avenue in Westwood - each row has the starting and ending latitude/longitude of a street segment that makes up a part of the overall street:

34.0671191, -118.4379955 34.0670930,-118.4377728
34.0670930, -118.4377728 34.0670621,-118.4376356
34.0670621, -118.4376356 34.0669753,-118.4374785
34.0669753, -118.4374785 34.0668906,-118.4373663
34.0668906, -118.4373663 34.0667584,-118.4372616
34.0667584, -118.4372616 34.0660314,-118.4369524
34.0660314, -118.4369524 34.0658228,-118.4368552
34.0658228, -118.4368552 34.0656493,-118.4367430
34.0656493, -118.4367430 34.0654861,-118.4365909
34.0654861, -118.4365909 34.0653477,-118.4363665
34.0653477, -118.4363665 34.0652111,-118.4359814
34.0652111, -118.4359814 34.0651391,-118.4356096

Notice that the ending lat/long of each segment is the same as the starting lat/long of the following segment, resulting in a contiguous street. And here's what Glenmont Ave looks like visually:



Let's consider the first segment in our list, which is highlighted above in red and blue:

34.0671191, -118.4379955 34.0670930,-118.4377728

If you look up 34.0671191, -118.4379955 in Google Maps (just type in these coordinates into the Google Maps search bar), you'll see that this is the location of the intersection of Malcolm Ave and Glenmont Ave (in the upper-left corner of the map). And if you look up 34.0670930,-118.4377728, you'll see that this refers to a spot maybe 100 feet down and right from Malcolm Ave, at the point at which Glenmont Ave. begins to curve just bit. Notice that the second segment for Glenmont Ave:

34.0670930, -118.4377728 34.0670621,-118.4376356

is directly connected to the first segment - the ending latitude/longitude of the first segment (in blue) is exactly the same as the starting latitude/longitude of the second segment (in green). In this manner, OSM can represent a long curvy street by stitching together multiple connecting line segments. (By the way, if you're not familiar with the latitude/longitude system, don't worry about it - for the purposes of this project, just assume that these are x,y points on a 2D grid.)

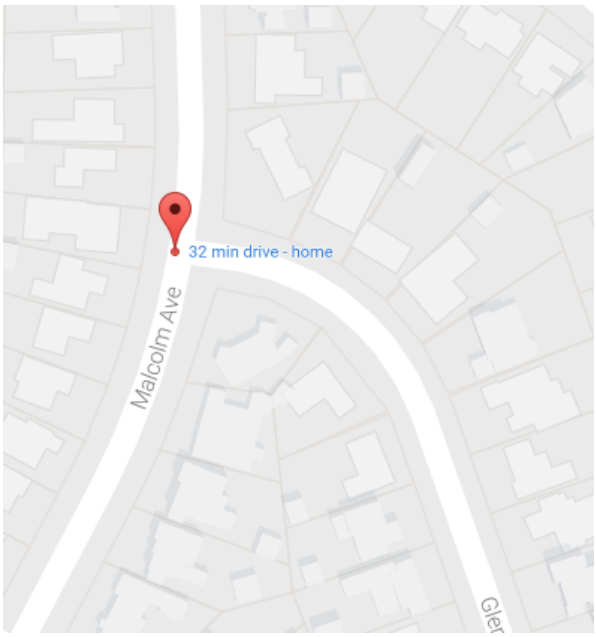
Now let's look at OSM's data for Malcolm Ave, which as we see in the map above intersects with Glenmont Ave. Here are a limited subset of the line segments that make up this much longer street:

...
34.0679593, -118.4379825 34.0676614,-118.4379719
34.0676614, -118.4379719 34.0673693,-118.4379684
34.0673693, -118.4379684 34.0671191,-118.4379955
34.0671191, -118.4379955 34.0668172,-118.4380882
34.0668172, -118.4380882 34.0665572,-118.4382046
34.0665572, -118.4382046 34.0660665,-118.4385079
34.0660665, -118.4385079 34.0654874,-118.4388836
...

You'll notice the highlighted line segment in the middle of the list. This line segment begins at coordinate

34.0671191, -118.4379955

which just happens to be the point of intersection of Glenmont and Malcolm, and was the first lat/long amongst the segments we showed you above for Glenmont Ave. We can thus see that these two streets intersect at this point!



So, you can imagine that given a starting geolocation (e.g., 34.0617768, -118.4466596 for 1031 Broxton Ave) and an ending geolocation (e.g., 34.0765967, -118.4196219 for The Maltz Park, a park in Beverly Hills), and given all of the street coordinates for all of the street segments in the OSM database, you should be able to find a contiguous chain of segments from the starting

location to the ending location, and then present this route to the user. Each segment that is part of this route will have its starting latitude, longitude matching the ending latitude, longitude of the previous segment.

And that's the goal of this project - to find a (near) optimal route from some starting coordinate to some ending coordinate!

Right now, you're probably thinking "There are thousands of streets in LA alone, and tens of thousands of street segments, how the heck am I supposed to sift through all that data to find a viable route?" Well, believe it or not, it's much easier than you think! And by the end of CS32, you'll have build your own turn-by-turn navigation system.

Street Map Data File

We will provide you with a simple data file (called *mapdata.txt)* that contains limited street map data for the Westwood, West Los Angeles, West Hollywood, Brentwood, and Santa Monica areas. This data file has a simplified format and was derived from OSM's more complicated XML-format data files. Our mapdata.txt file basically has data on thousands of individual street segments, which together make up the entire map. The file also holds the location of a number of popular attractions (e.g., In N Out Burger, Engineering IV) that your program will be responsible for navigating to/from. Here's an entry for a particular street segment of Gayley Ave. from the *mapdata.txt* file:

Gayley Avenue
34.0602175, -118.4464952 34.0597400,-118.4460477
3
Iso Fusion Café|34.0600264, -118.4460993
Native Foods Café|34.0599185, -118.4460044
Novel Cafe Westwood|34.0600033, -118.4465424

The first line of each segment holds the name of the street that this segment is associated with. In this case, this street segment is part of Gayley Avenue.

The second line holds the starting and ending geo-coordinates of the street segment in latitude, longitude format.

The third line specifies a count, C, of how many total attractions there are on this particular street segment.

Finally, there are C lines, one for each attraction found on this particular segment, detailing the name and geo-coordinates of the attraction separated by a pipe | character. Note: C's value

may be zero if there are no attractions on the segment. An attraction may be a place of business (e.g., “Iso Fusion Café”) or a street address (e.g., “1031 Gayley Ave”).

For example, the above data would represent the highlighted street segment below. You can see the attractions (Iso Fusion Cafe, Native Foods Cafe, and Novel Cafe) all represented on the map:

Here’s a slightly longer example from our map data file:

...
Glendon Avenue
34.0591340, -118.4426546 34.0589680,-118.4424895
0
Glendon Avenue
34.0589680, -118.4424895 34.0582358,-118.4421816
1
Pierce Brothers Westwood Village Memorial Park|34.0587141, -118.4418438
Glendon Avenue
34.0582358, -118.4421816 34.0572000,-118.4417620
1
CVS|34.0575488, -118.4423488
Wellworth Avenue
34.0575140, -118.4405712 34.0572000,-118.4417620

0
…

Notice how the first Glendon Avenue segment has an ending geo-coordinate that matches the second Glendon Avenue segment's starting geo coordinate (34.0589680,-118.4424895). Further, notice that the second Glendon Avenue segment has an ending geo-coordinate that matches the starting geo-coordinate for the third Glendon Avenue segment (34.0582358, -118.4421816). So these three segments are effectively chained together by their start/end coordinates. Now consider the Wellworth Avenue segment. Its ending coordinate (34.0572000,-118.4417620) is the same as the ending geo-coordinate of the third Glendon Avenue segment. So this means that this location defines an intersection between Glendon Avenue and Wellworth Avenue. And in fact, if you look this up in Google maps, this is what you'll see:

So now you know how our mapping data is encoded. And hopefully you're beginning to see that if you have some clever data structures, given any geo-coordinate, you can determine all segments that start or end at that point. You could then follow each such segment to its other end, and figure out what segments it's connected to, and so on.

What Do You Need to Do?

For this project you will create five classes (each will be described below in more detail):

1. You will create a class template *MyMap* which works much like the C++ STL *map* and which must use a binary search tree as its data structure. This class template will hold associations between an arbitrary type of key (e.g., a string containing an attraction

name like “Barney’s Beanery”) with an arbitrary type of value (e.g., a latitude/longitude where that attraction is located).
2. You will create a class *MapLoaderImpl* that is used to load up the data from the *mapdata.txt* file that we provide, so the data can be used by your program.
3. You will create a class *AttractionMapperImpl* that can be used to look up an attraction by name, e.g. “Mongol BBQ”, and will return that attraction’s geo-coordinate, if it was found in our data file.
4. You will create a class *SegmentMapperImpl* that can be used to look up a geo-coordinate (e.g., a latitude/longitude) and will return all segment(s) that are associated with that coordinate. That is, it will return all segments that either start at the specified geo-coordinate, end at the specified geo-coordinate, or have an attraction with the specified geo-coordinate located on the segment.
5. You will build a class called *NavImpl* that allows the user to specify a starting attraction (e.g., “Mongol BBQ”), an ending attraction (e.g., “Getty Conservation Institute”), and will then return a vector of turn-by-turn directions required to get from the starting point to the ending point.

What Will We Provide?

We’ll provide you with a header file named *provided.h* which you **must not** modify. It defines:

- A *GeoCoord* struct that you can use to hold a particular latitude/longitude, and a *GeoSegment* struct that defines a segment consisting of starting and ending GeoCoords.
- Functions to compute the distance between two GeoCoords, the angle of a GeoSegment, and the angle between two GeoSegments (i.e., street segments).
- A *StreetSegment* struct that holds details of a street segment loaded from a map data file: the name of the street segment, its starting and ending geocoordinates, and a vector of attractions on that segment.
- A *NavSegment* class. The Navigator’s navigate() method returns its routing directions as a sequence of these NavSegments. Each NavSegment holds data on either (a) one segment of the route (e.g., a street name, and the segment’s starting and ending geo-coordinates), or (b) a turn instruction, detailing a turn that must be made between two segments in the route.
- *MapLoader, AttractionMapper, SegmentMapper,* and *Navigator* classes. The code you write will implement these classes.

We’ll provide a simple *main.cpp* file that brings your entire program together and lets you test it. You **MUST not** modify this file, as you will not turn it in with your solution. (Of course, you can modify it during development, but the program you turn in must work correctly with the main.cpp that we provided.)

We also provide you with two data files:

- *mapdata.txt*: Contains all of the mapping data (latitudes and longitudes for each street, street names, etc.) that you have to process in your program.
- *validlocs.txt*: Contains a list of attraction names/addresses (e.g., “Engineering IV” or “Barney’s Beanery”) extracted from mapdata.txt that you can route from or route to when testing your program.

Our Test Driver

If you compile your code with our main.cpp file, you can use it to test your completed classes. Our main.cpp file implements a command-line interface, meaning that if you open a Windows/MacOS Command Shell (e.g., by typing “cmd.exe” in the Windows start box in the bottom-left corner of the screen, or by running the Terminal app in MacOS), and switch to the directory that holds your compiled executable file, you can run our test harness code.

From the command line, you can run the test harness as follows:

```
C:\PATH\TO\YOUR\CODE> BruinNav.exe c:\path\to\the\mapdata.txt "start location name" "end location name”
```

So, for example, if you wanted our test driver to test out your navigation code to get directions from 1031 Broxton Ave. to The Maltz Park, you’d write:

```
C:\cs32\p4> BruinNav.exe c:\cs32\p4\mapdata.txt "1031 Broxton Ave." “The Maltz Park”
```

Our test program will then run, take the inputs you passed on the command line (e.g., The Maltz Park) and pass them to your classes so they can load the appropriate map data and generate a route. The test program will then take the results from your classes (e.g., routing instructions passed back in a vector), and print them to the screen like this:

```
You are starting at: 1031 Broxton Avenue
Proceed 0.09 miles southeast on Broxton Avenue
Turn left onto Kinross Avenue
Proceed 0.07 miles east on Kinross Avenue
Turn right onto Glendon Avenue
Proceed 0.08 miles southeast on Glendon Avenue
Turn left onto Lindbrook Drive
Proceed 0.93 miles east on Lindbrook Drive
Turn right onto Holmby Avenue
Proceed 0.08 miles southeast on Holmby Avenue
Turn left onto Wilshire Boulevard
Proceed 0.92 miles east on Wilshire Boulevard
Turn left onto Whittier Drive
Proceed 0.74 miles north on Whittier Drive
```

```
You have reached your destination: The Maltz Park
Total travel distance: 2.9 miles
```

You can then manually check your solution using Google maps!

Details: The Classes You Must Write

You must write correct versions of the following classes to obtain full credit on this project. Your classes must work correctly with our provided classes (with no modifications to our provided classes).

MyMap

You must implement a template class named *MyMap* that, like an STL map, lets a client associate items of a key type with items of a (usually different) value type, with the ability to look up items by key. For example, a MyMap object associating students' names with their GPAs would have string as the key type and double as the value type. Your implementation **must** use a binary search tree.

Here's an example of how you might use MyMap:

```
void foo()
{
      MyMap<string,double> nameToGPA;  // maps student name to GPA

      // add new items to the binary search tree-based map
      nameToGPA.associate("Carey", 3.5);  // Carey has a 3.5 GPA
      nameToGPA.associate("David", 3.99); // David beat Carey
      nameToGPA.associate("Abe", 3.2);    // Abe has a 3.2 GPA

      double* davidsGPA = nameToGPA.find("David");
      if (davidsGPA != nullptr)
          *davidsGPA = 1.5;   // after a re-grade of David's exam

      nameToGPA.associate("Carey", 4.0);  // Carey deserves a 4.0
                                          // replaces old 3.5 GPA

      double* lindasGPA = nameToGPA.find("Linda");
      if (lindasGPA == nullptr)
          cout << "Linda is not in the roster!" << endl;
      else
          cout << "Linda's GPA is: " << *lindasGPA << endl;
```

```
}
```

Your implementation **must** have the following interface:

```
template <typename KeyType, typename ValueType>
class MyMap
{
public:
	MyMap();			// constructor
	~MyMap();			// destructor; deletes all of the tree's nodes
	void clear();		// deletes all of the trees nodes producing an empty tree
	int size() const;	// return the number of associations in the map

	// The associate method associates one item (key) with another (value).
	// If no association currently exists with that key, this method inserts
	// a new association into the tree with that key/value pair.  If there is
	// already an association with that key in the tree, then the item
	// associated with that key is replaced by the second parameter (value).
	// Thus, the tree contains no duplicate keys.
	void associate(const KeyType& key, const ValueType& value);

	// If no association exists with the given key, return nullptr; otherwise,
	// return a pointer to the value associated with that key.  This pointer can be
	// used to examine that value, and if the map is allowed to be modified, to
	// modify that value directly within the map (the second overload enables
	// this).  Using a little C++ magic, we have implemented it in terms of the
	// first overload, which you must implement.
	const ValueType* find(const KeyType& key) const;
	ValueType* find(const KeyType& key);
};
```

Requirements for MyMap

Here are the requirements for your MyMap class:

1. You **must** implement your own binary search tree in your *MyMap* class (i.e., define your own *Node* struct/class, maintain a root/head pointer, etc). You may assume that the key type of any instantiation of the MyMap template class has appropriate comparison operators (<, <=, >, >=, ==, and !=) defined for it (certainly ints and strings do).
2. Your *MyMap* class **must** be a template class, to enable a client to map one type of item to any other type of item, e.g., a name (string) to a GPA (double), or a name (string) to a collection of the person's test scores (a vector of ints).
3. Your *MyMap* class **must** use the public interface documented above. You may add only private members to this class; you must **not** add other public members to *MyMap*.

4. Your *MyMap* class does not need to implement deletion of an individual association (unless you really want to) and does not need to attempt to keep the tree balanced (unless you're masochistic).
5. If a user of your class associates the same key twice (e.g., "David" to 3.99, then "David" to 1.5), the second association must overwrite the first one (i.e., "David" will no longer be associated with 3.99, but will henceforth be associated with 1.5). There **must** be at most one mapping for any key.
6. *MyMap* objects do not need to be copied or assigned. To prevent incorrect copying and assignment of *MyMap* objects, these methods can be declared to be deleted (C++11) or declared private and left unimplemented (pre-C++11).
7. Your member functions **MUST not** write anything out to *cout*. They may write to *cerr* if you like (to help you with debugging).

MapLoader

The MapLoader class is used to load data from our provided mapdata.txt file. Here is the required public interface of the MapLoader class:

```
class MapLoader
{
public:
      MapLoader();
      ~MapLoader();
      bool load(std::string mapFile);
      size_t getNumSegments() const;
      bool getSegment(size_t segNum, StreetSegment& seg) const;
};
```

After constructing a MapLoader object, the client calls the object's load() method, passing in the name of a map data file. Your load() method must load all of the data from the specified map data file into a container of StreetSegments (you may use a vector or other dynamic array to hold your data). You'll need to ensure that you've loaded every street segment from the file, each into its own StreetSegment object. The format of the map data file is specified above in the Street Map Data File section.

You'll have to use the *ifstream* class to open our data file and read the data line by line from this file into your MapLoader object (see the File I/O writeup on the class web site). The load() method must return true if the data was loaded successfully, and false otherwise. You may assume that the data in the map data file is formatted correctly as detailed in this specification, so you don't have to check for errors in its format.

Once you have loaded the data, a call to getNumSegments() must return the total number of segments loaded. Otherwise, a call to getNumSegments() must return 0.

A call to getSegment() must retrieve the StreetSegment associated with the specified segment number (segNum must be between 0 and getNumSegments()-1), and place it in the seg reference parameter. If the specified segment number is invalid (out of bounds), then the method must return false, leaving seg unchaged. Otherwise, it must return true and fill in the seg parameter.

The filled-in seg parameter must be completely filled in if the function is successful, containing proper field values for streetName and segment, and a properly filled in attractions vector.

To ensure that you do not change the interface to the MapLoader class in any way, we will implement that class for you. But don't get your hopes up that we're doing any significant work for you here: Our implementation is to simply give MapLoader just one private data member, a pointer to a MapLoaderImpl object (which you can define however you want in MapLoader.cpp). The member functions of MapLoader simply delegate their work to functions in MapLoaderImpl.[1] You still have to do the work of implementing those functions.

Other than MapLoader.cpp, no source file that you turn in may contain the name MapLoaderImpl. Thus, your other classes must not directly instantiate or even mention MapLoaderImpl in their code. They may use the MapLoader class that we provide (which indirectly uses your MapLoaderImpl class).

Requirements for MapLoader

Here are the requirements for your MapLoaderImpl class that implements the MapLoader functionality:

1. It **must** adhere to the specification above.
2. It must **not** access any other Impl classes that you write.
3. It must **not** use any STL associative containers (i.e., map, multimap, set, multiset, or the unordered_ versions of those classes). It **may** use the STL vector, list, stack, queue, and priority_queue classes if you wish.
4. It must **not** write anything to *cout*. It may write to *cerr* if you with (to help you with debugging).
5. If there are N lines in the input mapping data file, then load() must run in O(N) time, and getNumSegments() and getSegment() must run in O(1) time.

[1] This is an example of what is called the pimpl idiom (from "pointer-to-implementation").

AttractionMapper

The AttractionMapper class is used to look up an attraction name (e.g., “Engineering VI” or “1049 Gayley Avenue”) and find the GeoCoord associated with that attraction name.

```
class AttractionMapper
{
public:
      AttractionMapper();
      ~AttractionMapper();
      void init(const MapLoader& ml);
      bool getGeoCoord(std::string attraction, GeoCoord& gc) const;
};
```

After constructing an AttractionMapper object, a client will first call the init() method, passing in a MapLoader object (which has already been loaded up with map data). The init() method uses that object to construct an efficient data structure that allows the getGeoCoord() method to quickly find the GeoCoord that is associated with the specified attraction name. If getGeoCoord() finds the attraction, it sets the gc parameter to the corresponding GeoCoord and returns true; otherwise, it leaves gc unchanged and returns false. The function is **case insensitive**, so “barneY’s BEANery” and “barney’s beanery” would match an attraction name of “Barney’s Beanery” as found in the mapdata.txt file. This function must run in O(log N) time on average, where N is the number of attraction-to-geolocation mappings in the data structure.

Here’s an example of how you might use this class:

```
#include "provided.h"  // defines class AttractionMapper

void example(const MapLoader& ml)
{
      AttractionMapper am;
      am.init(ml);        // let our object build its internal data structures
                          // by iterating thru all segments from the MapLoader object
      GeoCoord fillMe;
      string attraction = "The Coffee Bean & Tea Leaf";

      bool found = am.getGeoCoord(attraction, fillMe);
      if ( ! found)
      {
         cout << "No geolocation found for " << attraction << endl;
         return;
      }

      cout << "The location of " <<  attraction << " is " <<
              << fillMe.sLatitude << ", " << fillMe.sLongitude << endl;
```

```
}
```

As with the other classes you must write, the real work will be implementing the auxiliary class AttractionMapperImpl in AttractionMapper.cpp. **Other than AttractionMapper.cpp, no source file that you turn in may contain the name AttractionMapperImpl.** Thus, your other classes must not directly instantiate or even mention AttractionMapperImpl in their code. They may use the AttractionMapper class that we provide (which indirectly uses your AttractionMapperImpl class).

Requirements for AttractionMapper

Here are the requirements for your AttractionMapperImpl class:

1. It **must** adhere to the specification above.
2. It must **not** access any other Impl classes that you write.
3. It **must** use your MyMap class template, and must **not** use **any** STL containers — no map, no vector, no list, etc.
4. It must **not** write anything to *cout*. It may write to *cerr* if you with (to help you with debugging).
5. If there are N total street segments in the input mapping data, and A total attractions dispersed throughout the streets, then your init() method must run in O(N+A*log(A)) time on average, and your getGeoCoord() must run in O(log(A)) time on average. You may assume that the map data being loaded is randomly ordered.

SegmentMapper

The SegmentMapper class is used to look up a geocoordinate (a latitude/longitude pair) and find the one or more StreetSegments associated with that coordinate. A StreetSegment is associated with a geocoordinate if:

1. The StreetSegment starts at that geocoordinate
2. The StreetSegment ends at that geocoordinate
3. The StreetSegment has an attraction on it with that geocoordinate

```
class SegmentMapper
{
public:
      SegmentMapper();
      ~SegmentMapper();
      void init(const MapLoader& ml);
      std::vector<StreetSegment> getSegments(const GeoCoord& gc) const;
};
```

After constructing a SegmentMapper object, a client will first call the init() method, passing in a MapLoader object (which has already been loaded up with map data). The init() method uses that object to construct an efficient data structure that allows the getSegments() method to quickly find all StreetSegments that are associated with the specified geocoordinate. The getSegments() method returns a vector containing all those StreetSegment; if there are no StreetSegments associated with that geocoordinate, the returned vector will be empty.

Here's an example of how you might use this class:

```
#include "provided.h"  // defines class SegmentMapper

void example(const MapLoader& ml)
{
       SegmentMapper sm;
       sm.init(ml);        // let our object build its internal data structures
                           // by iterating thru all segments from the MapLoader object

       GeoCoord lookMeUp("34.0572000", "-118.4417620");

       std::vector<StreetSegment> vecOfAssociatedSegs(sm.getSegments(lookMeUp));
       if (vecOfAssociatedSegs.empty())
       {
          cout << "Error - no segments found matching this coordinate\n";
          return;
       }

       cout << "Here are all the segments associated with your coordinate:" << endl;

       for (auto s: vecOfAssociatedSegs)
       {
          cout << "Segment's street: " << s.streetName << endl;
          cout << "Segment's start lat/long: " << s.segment.start.sLatitude << ", " <<
                                                  s.segment.start.sLongitude << endl;
          cout << "Segment's end lat/long: " << s.segment.end.sLatitude << ", " <<
                                                  s.segment.end.sLongitude << endl;
          cout << "This segment has " << s.attractionsOnThisSegment.size() <<
                    " attractions on it." << endl;
       }
}
```

As with the other classes you must write, the real work will be implementing the auxiliary class SegmentMapperImpl in SegmentMapper.cpp. **Other than SegmentMapper.cpp, no source file that you turn in may contain the name SegmentMapperImpl.** Thus, your other classes must not directly instantiate or even mention SegmentMapperImpl in their code. They may use the SegmentMapper class that we provide (which indirectly uses your SegmentMapperImpl class).

Requirements for SegmentMapper

Here are the requirements for your SegmentMapperImpl class:

1. It **must** adhere to the specification above.
2. It must **not** access any other Impl classes that you write.
3. It **must** use your MyMap class template, and must **not** use any STL associative containers (i.e., map, multimap, set, multiset, or the unordered_ versions of those classes). It **may** use the STL vector, list, stack, queue, and priority_queue classes if you wish.
4. It must **not** write anything to *cout*. It may write to *cerr* if you with (to help you with debugging).
5. If there are N total street segments in the input mapping data, and A total attractions dispersed throughout the streets, then your init() method must run in O((N+A)*log(N+A)) time on average, and your getSegments() must run in O(log(N+A)) time on average. You may assume that the map data being loaded is randomly ordered.

Navigator

The Navigator class is responsible for computing an efficient route from a source attraction to a destination attraction, if one exists. It must use the AttractionMapper, SegmentMapper, and MapLoader classes to do so.

```
class Navigator
{
public:
      Navigator();
      ~Navigator();
      bool loadMapData(std::string mapFile);
      NavResult navigate(std::string start, std::string end,
                         std::vector<NavSegment>& directions) const;
};
```

After constructing a Navigator object, the client calls the object's loadMapData() method, passing in the name of a map data file. Your loadMapData() method must load all required data and initialize all internal data structures (including perhaps an AttractionMapper and a SegmentMapper). The loadMapData() method must return true if the data was loaded successfully, and false otherwise. You may assume that the data in the map data file is formatted correctly as detailed in this specification, so you don't have to check for errors in its format.

After having loaded the map data, a client can then call the navigate() method, passing in:

- A starting attraction name or address, e.g. “Westwood Sporting Goods” or “1031 Broxton Avenue”
- An ending attraction name or address, e.g. “Easton Softball Stadium”
- A vector to be filled in by the navigate() function if it finds a valid route from the starting attraction to the ending attraction. If the vector passed in is not empty, then the navigate() method must clear it before filling it in with results.

Note: The *validlocs.txt* file that we provide contains a list of attraction names/addresses extracted from mapdata.txt that you can use when testing your program.

Your navigate() method must return one of the following codes, depending on the result:

- **NAV_SUCCESS**: A path was found from the source to the destination.
- **NAV_BAD_SOURCE**: The source attraction or street address that was passed in was not found in our data file, and therefore the system can’t route from it.
- **NAV_BAD_DESTINATION**: The destination attraction or street address that was passed in was not found in our data file, and therefore the system can’t route to it.
- **NAV_NO_ROUTE**: No route was found linking the source to the destination address.

If your navigate() function can find a valid, connecting set of segments from the source attraction/address to the destination attraction/address, it must fill the directions vector parameter with a sequence of NavSegments that represent turn-by turn directions. Note: A NavSegment is different than a StreetSegment and a GeoSegment. NavSegments are used to convey turn-by-turn directions to the user. There are two types of NavSegments: Proceed-style NavSegments and Turn-style NavSegments.

A **Proceed-style NavSegment** specifies a starting latitude/longitude of the segment, an ending latitude/longitude of the segment, the compass direction of travel from start to end (e.g., north, southwest, etc.), the distance in miles to travel, and the name of the street being travelled on (e.g., Broxton Avenue). You can determine the distance by using distanceEarthMiles() and the compass direction by using angleOfLine(), both functions being defined in the provided.h file[2]:

- 0 degrees <= travelAngle <= 22.5 degrees: east
- 22.5 degrees < travelAngle <= 67.5 degrees: northeast
- 67.5 degrees < travelAngle <= 112.5 degrees: north
- 112.5 degrees < travelAngle <= 157.5 degrees: northwest
- 157.5 degrees < travelAngle <= 202.5 degrees: west
- 202.5 degrees < travelAngle <= 247.5 degrees: southwest
- 247.5 degrees < travelAngle <= 292.5 degrees: south
- 292.5 degrees < travelAngle <= 337.5 degrees: southeast
- 337.5 degrees < travelAngle < 360 degrees: east

[2] Note: Unlike typical compass angles, where 0 degrees is due north, our angleOfLine() function returns 0 degrees facing due east.

A **Turn-style NavSegment** specifies a direction to turn (left or right), and the name of the street that's being turned onto. A turn of less than 180 degrees from one segment onto another indicates a left turn. A turn of 180 degrees or more indicates a right turn.

Your outputted direction vector must have exactly one Proceed-style NavSegment for each GeoSegment segment travelled over.

Your direction vector must have exactly one Turn-style NavSegment when a transition is made from a first GeoSegment associated with street A to a second GeoSegment associated with street B, where A != B. You must not have a turn-style NavSegment when proceeding between two segments with the *same* street name (i.e., where A == B).

To illustrate how the navigate() method must fill in its directions vector, let's go through a simple example. Let's assume that we're navigating between **"1061 Broxton Avenue"** and **"Headlines"** (on Kinross Avenue). Here's a map that shows the route, along with the various segments as found in the mapdata.txt file:

And here's the raw street segment data from our mapdata.txt file (We've added segment #s in parenthesis to make referencing the segments easier):

```
Broxton Avenue (Segment #1)
34.0620596, -118.4467237 34.0613323,-118.4461140
9
1031 Broxton Avenue|34.0617768, -118.4466596
1037 Broxton Avenue|34.0615332, -118.4468449
1045 Broxton Avenue|34.0616887, -118.4465843
1055 Broxton Avenue|34.0612865, -118.4466416
1061 Broxton Avenue|34.0613269, -118.4462765
Ami Sushi|34.0614911, -118.4464410
Barney's Beanery|34.0617224, -118.4466561
Five Guys|34.0613946, -118.4463597
Regent|34.0615961, -118.4465521
```

```
Broxton Avenue (Segment #2)
34.0613323, -118.4461140 34.0609137,-118.4457707
2
1067 Broxton Avenue|34.0612157, -118.4461814
1073 Broxton Avenue|34.0611019, -118.4460843
Broxton Avenue (Segment #3)
34.0608001, -118.4457307 34.0607063,-118.4457055
0
Kinross Avenue (Segment #4)
34.0607063, -118.4457055 34.0604893,-118.4460593
0
Kinross Avenue (Segment #5)
34.0604893, -118.4460593 34.0602175,-118.4464952
3
10925 Kinross Avenue|34.0606777, -118.4466919
Bel Air Camera|34.0604846, -118.4464268
Headlines!|34.0602020, -118.4462382
```

For the moment, let's gloss over how your navigate() method actually computes an efficient route (you may assume that there's some way to do this), and just consider what a valid output directions vector might look like upon the method's successful completion.

The first entry in all directions vectors must contain a proceed-style NavSegment whose start GeoCoord is the latitude/longitude of the source attraction, and whose end GeoCoord is the latitude/longitude of one of the ends of the GeoSegment containing that attraction, where the navigation system wants the user to travel to next. The direction travelled is determined by computing the angle between the starting and ending coordinates of the NavSegment, and using this to determine the compass direction (e.g., east, southeast), etc. according to the formula in the section above.

So, in the above example of navigating from 1061 Broxton to Headlines!, the first NavSegment's value would be:

directions[0]:
- type: PROCEED
- start: 34.0613269, -118.4462765
- end: 34.0613323,-118.4461140
- direction: "east"
- distance: 0.00930891
- street: "Broxton Avenue"

You'll notice that the start coordinate is the geolocation of 1061 Broxton Avenue (see the attractions listed under the data for Segment #1 above), and the end coordinate is the

geolocation of the south end of Segment #1, since the navigation system wants us to travel east[3] down Broxton.

The second NavSegment takes us from the end of Segment #1 (34.0613323,-118.4461140… which also happens to be the start of Segment #2) to the end of Segment #2:

directions[1]:
type: PROCEED
start: 34.0613323,-118.4461140
end: 34.0609137,-118.4457707
direction: “southeast”
distance: 0.0349664
street: “Broxton Avenue”

Notice that we didn’t have a Turn-style NavSegment between directions[0] and directions[1], since we’re proceeding down the same street, Broxton Avenue, rather than turning to a new street.

The third NavSegment takes us from the end of Segment #2 (34.0609137,-118.4457707… which also happens to be the start of Segment #3) to the end of Segment #3:

directions[2]:
type: PROCEED
start: 34.0609137,-118.4457707
end: 34.0607063,-118.4457055
direction: “southeast”
distance: 0.014808
street: “Broxton Avenue”

At this point, we’re turning onto a new street. So we must have a Turn NavSegment next in our direction vector:

directions[3]:
type: TURN
direction: “right”
street: “Kinross Avenue”

The fifth NavSegment takes us from the end of Segment #3 (34.0607063,-118.4457055… which also happens to be the start of Segment #4) to the end of Segment #4:

[3] It would appear that we’re traveling southeast down Broxton, but since the start coordinate is 1061 Broxton, which is in the middle of the street, the path to the south end of Segment #1 actually takes us more eastward than southeast.

directions[4]:
    type: PROCEED
    start: 34.0607063,-118.4457055
    end: 34.0604893,-118.4460593
    direction: “southwest”
    distance: 0.0251977
    street: “Kinross Avenue”

The sixth NavSegment takes us from the end of Segment #4 (34.0604893,-118.4460593… which also happens to be the start of Segment #5) to the end destination of Headlines! (34.0602020, -118.4462382) which is located on Segment #5:

directions[5]:
    type: PROCEED
    start: 34.0604893,-118.4460593
    end: 34.0602020,-118.4462382
    direction: “southwest”
    distance: 0.0223362
    street: “Kinross Avenue”

And now we have reached our destination!

A few things you should notice:

1. The first NavSegment must always have the source attraction as its start coordinate, and one of the ends of the GeoSegment that holds the source attraction as its end coordinate, **with one exception**: If the source attraction and destination attraction are on the same GeoSegment (e.g., both 1031 Broxton and 1061 Broxton are on Segment #1), then the start coordinate will be that of the source attraction, and the end coordinate will be that of the destination attraction (and there will be only one NavSegment in your direction vector).
2. If the start attraction and end attraction are on different GeoSegments, then we will have one or more NavSegments following the first NavSegment:
    a. If two adjacent GeoSegments are for the same street (like Segment #1 and Segment #2, or Segment #2 and Segment #3), then you simply add successive Proceed-style NavSegments in your directions vector.
    b. If two adjacent GeoSegments are for a different street (like Segment #3 and Segment #4), then you must add a Turn-style NavSegment to your direction vector in between the two streets, indicating that a turn must be made between the two streets.
3. Your final NavSegment must always have the destination attraction as its end coordinate, and one of the ends of the GeoSegment that holds the destination attraction as its start coordinate (unless of course, both attractions are on the same GeoSegment, as explained in #1 above).

4. For all Proceed NavSegments for the same street, except for the first and last, the j+1st NavSegment's start coordinate must be the same as the jth NavSegment's end coordinate, unless there is a Turn NavSegment in between two Proceed NavSegments, in which the jth NavSegment's end coordinate must be the same as the j+2nd NavSegment's start coordinate, with a Turn NavSegment in position j+1.

As with the other classes you must write, the real work will be implementing the auxiliary class NavigatorImpl in Navigator.cpp. **Other than Navigator.cpp, no source file that you turn in may contain the name NavigatorImpl.** Thus, your other classes must not directly instantiate or even mention NavigatorImpl in their code. They may use the Navigator class that we provide (which indirectly uses your NavigatorImpl class).

Requirements for Navigator

Here are the requirements for your NavImpl class:

1. It **must** adhere to the specification above.
2. It must **not** access any other Impl classes that you write.
3. It must **not** use any STL associative containers (i.e., map, multimap, set, multiset, or the unordered_ versions of those classes). It **may** use the STL vector, list, stack, queue, and priority_queue classes if you wish. It **may** use your MyMap class template.
4. It must **not** write anything to *cout*. It may write to *cerr* if you with (to help you with debugging).
5. Assuming there are N total segments and A total attractions in our mapping data, your navigate() method must run in O((A+N)*log(A+N)) time. However, if you implement your route-finding algorithm efficiently, it should generally run in far less time.

How to Implement a Route-finding Algorithm

Right now you're probably thinking: "It's got to be rocket science to compute an optimal route between two coordinates on a map… only a company as awesome as Google could do that :)". But in fact, nothing could be further from the truth! It's possible to implement an optimal routing algorithm in just a few hundred lines of code (or less!) using an algorithm known as A*. You're welcome to use A* if you like, but if you're a little intimidated by this, why not use an algorithm like the queue-based maze searching algorithm you implemented in your homework?

Of course, there are a few differences between maze-searching and geo-navigation:

1. In your original queue-based maze searching algorithm, you enqueued integer-valued x,y coordinates, whereas in this project you're enqueuing real-valued latitude, longitude coordinates.
2. In your original maze searching algorithm, you "dropped breadcrumbs" in your maze array to prevent your algorithm from visiting the same square more than once, whereas

in this project, there's no 2D array that you can use to track whether you've visited a square, so you'll have to figure out some other way to prevent your algorithm from re-visiting the same coordinates over and over.

3. In the original maze-searching algorithm, you could determine adjacent squares of the maze to explore through simple arithmetic - if you were at position (x,y) of the maze, you knew that the adjacent maze locations were (x-1,y), (x+1,y), (x,y-1), and (x,y+1). In this project, you're going to have to leverage the SegmentMapper class to locate adjacent coordinates, and the AttractionMapper to obtain your start and destination coordinates.
4. In the original maze-searching algorithm, you just had to determine if the maze was solvable and return a boolean result (true or false). But for this project, you'll have to actually return back a full vector of segments to provide turn-by-turn directions.

But, with just a few changes, you should be able to adapt your queue-based maze searching algorithm to one that does street navigation! Of course, you're still going to have to figure out how to return the whole route (segment-by-segment directions) back to the user. You can't just search and verify that a route exists like in your maze-searching homework - you have to return each segment that makes up that route!

How might you track the complete segment-by-segment route so you can return it back to the user? Well, one way to do so would be to maintain a map (using your templated MyMap class) that associates a given geo-coordinate G to the previous geo-coordinate P in the route (e.g., we travelled **to** G directly **from** P). Let's call this map: *locationOfPreviousWayPoint*

Let's illustrate the approach with an example containing 4 segments:

- Segment A-B: Contains the starting location, A' (e.g., Barney's Beanery)
- Segment B-C: Connected to Segment A-B at geo-coordinate B
- Segment C-D: Connected to Segment B-C at geo-coordinate C, contains destination location D' (e.g., Engineering VI)

Let's assume that your queue-based search algorithm is searching for a path from A' to D'. Starting from point A', its algorithm might discover that A' is on segment A-B, and therefore enqueue points B and A (the two ends of the segment containing A') for exploration by the algorithm. At this point, the algorithm would add the following two associations to your map:

locationOfPreviousWayPoint[B] -> A'
locationOfPreviousWayPoint[A] -> A'

The first association indicates that we reached location B directly from location A'. The second association similarly indicates we reached location A directly from location A'.

Next, the algorithm might dequeue point B, and determine that it can reach geo-coordinate C from B. It would add C to its queue for later exploration. And, again, it would add the fact that it reached C from B to its waypoint map:

locationOfPreviousWayPoint[C] -> B

A bit later (after processing geo-coordinate A in the queue, which we'll omit for brevity), the algorithm might dequeue geo-coordinate C from the queue. From geo-coordinate C, the algorithm could then determine that attraction D' is on segment C-D, and that it has found the destination attraction. Again, it could add this fact to the map:

locationOfPreviousWayPoint[D'] -> C

So every time we reach a new waypoint (e.g., B or C or D'), the algorithm could add an entry to the *locationOfPreviousWayPoint* map that maps that waypoint to the geo-coordinate that we traveled *from* to get to that waypoint.

When we finally reach point D', our map might contain:

locationOfPreviousWayPoint[B] -> A'
locationOfPreviousWayPoint[A] -> A'
locationOfPreviousWayPoint[C] -> B
locationOfPreviousWayPoint[D'] -> C
...

So how can we use this map to reconstruct our route from A' to D'? Well, starting from our destination point - location D' - we can look up D' in the map to determine how we got there (from C). This tells us that our last segment in our navigation was (C,D'). We can then lookup point C in our map and determine that we got there from point B. This tells us that the next to last segment was (B,C). And so on. Eventually we'll reach point A', our starting point, allowing us to complete the first segment (A',B), and we'll have re-created the complete route. Each of these discovered segments can be added to a vector and then returned to the user.

Since, like the maze searching algorithm, your navigation algorithm must only visit each geo-coordinate once (otherwise it would potentially go in circles), you're guaranteed to have a single entry for each point in your *locationOfPreviousWayPoint* map for each geo-coordinate, enabling you to easily reconstruct the route. There should never be a case where the map has to associate a given waypoint with more than one previous coordinate. Cool, huh?

So, intuitively, what does your *overall* navigation algorithm do? Well, it basically moves out from the starting attraction in concentric growing rings. It starts by locating the start and end GeoCoords of the segment that holds our starting attraction and adds them to our queue. It then finds all segments associated with these first two GeoCoords and adds the other ends of their segments to our queue. It then finds all segments associated with these GeoCoords and adds the other ends of their segments to our queue. And so on, and so on. Eventually, either the algorithm stumbles upon the segment that holds our destination attraction, or it will work our way through the entire street map, completely empty out our queue, and realize that the

destination can't be reached. If and when the algorithm finds the ending attraction, it can then use the locationOfPreviousWayPoint map to trace a path back from the ending location to the starting location, following each geolocation it traversed back to the one just before it, and to the one before it, all the way to the start attraction.

Now if you think hard, you'll realize that this algorithm won't necessarily find the shortest path (in miles) from your source attraction to the destination attraction. It will, however, find the path with the fewest number of segments between your source to your destination and return it to you. But the path with the smaller number of segments won't necessarily result in the shortest/fastest path. Consider a case where there are three points: A, B and C all on a straight line. A is at position 0, B is at position 10 miles, and C is at position 100 miles. There are two sets of roads that can take us from point A to point B:

- A curvy road that has 20 segments that proceed directly from A to B, with a total distance of 10 miles.
- A straight road that has 1 segment that proceeds from A to C (100 miles), and then a second straight segment that proceeds from C back to B (90 miles).

Our naive queue-based algorithm would end up selecting the second option, even though it's far less efficient, because it requires fewer total hops/segments to reach the endpoint. This is suboptimal.

There are various ways to make your algorithm find a better/faster path, for instance using the A* algorithm. Or, you could do something really simple… For example, imagine that your algorithm is currently searching for a route and is at geo-coordinate X. Further, let's assume that X is connected to three outgoing segments (X,Y), (X,Z), and (X,Q), and that locations Y, Z and Q have not yet been visited.

Rather than just enqueuing Y, Z and Q into our queue in some arbitrary order, we could rank-order those three coordinates by their distance to our ultimate destination, and then insert each item into our queue in order of its increasing distance from the destination attraction. So if location Z is .1 miles away from our destination, location Y is 2.5 miles, and location Q is .6 miles, we might enqueue location Z first, Q second and Y third. This "heuristic" will increase (though not guarantee) the likelihood that we'll find the shortest path first. Use your creativity to improve upon the basic queue-based navigation algorithm, or look up A* - it's actually pretty simple to implement and will give an optimal result.

Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

1. In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project / Properties / Configuration Properties / General / Character Set
2. The entire project can be completed in under 500 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
3. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures? Plan before you program!
4. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
5. You must not modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
6. Your Impl classes (e.g., AttractionMapperImpl, NavigatorImpl) must **never** ***directly*** use your other Impl classes. They MUST use our provided wrapper classes instead:

INCORRECT:

```
class NavigatorImpl
{
…
   AttractionMapperImpl m_attractionMapper;   // BAD!
…
};
```

CORRECT:

```
class NavigatorImpl
{
…
   AttractionMapper m_attractionMapper;   // GOOD!
…
};
```

7. Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
8. You may use only those STL containers (e.g., vector, list) that are not forbidden by this spec. Use the MyMap class if you need a map, for example; do **not** use the STL *map* or *unordered_map* class.

9. Let *Whatever* represent MapLoader, AttractionMapper, SegmentMapper, and Navigator. Subject to the constraints we imposed (e.g., no changes to the public interface of the *Whatever* class, no mention of *Whatever*Impl in any file other than *Whatever*.cpp, no use of certain STL containers in your implementation), you're otherwise pretty much free to do whatever you want in *Whatever*.cpp as long as it's related to the support of only the *Whatever* implementation; for example, you may add members (even public ones) to the *Whatever*Impl class (but not the *Whatever* class, of course) and you may add non-member support functions (e.g., a custom comparison function for *sort()*).

If you don't think you'll be able to finish this project, then take some shortcuts. For example, if you can't get your MyMap class working, use the substitute MyMap class we provide so that you can proceed with implementing other classes, and go back to fixing your MyMap class later.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., SegmentMapperImpl), we will provide a correct version of that class and test it with the rest of your program (by changing our SegmentMapper class to use our version of the class instead of your version). If you implemented the rest of the program properly, it should work perfectly with our version of the SegmentMapperImpl class and we can give you credit for those parts of the project you completed (This is why we're using Impl classes and non-Impl classes).

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors under both g32 and either Visual Studio or clang++!

What to Turn In

You must turn in **six to eight** files. These six are required:

MyMap.h	Contains your BST map class template implementation
MapLoader.cpp	Contains your map loader implementation
AttractionMapper.cpp	Contains your attraction mapper implementation
SegmentMapper.cpp	Contains your segment mapper implementation
Navigator.cpp	Contains your navigation system implementation
report.doc, report.docx, or report.txt	Contains your report

These two are optional:

support.h	You may define support constants/classes/functions in these support files
support.cpp	and use them in your other source files

Use support.h and support.cpp if there are constants, class declarations, functions, and the like that you want to use in *more than one* of the other files. (If you wanted to use something in only

one file, then just put it in that file.) Use support.cpp only if you declare things in support.h that you want to implement in support.cpp.

You are to define your class declarations and all member function implementations directly within the specified .h and .cpp files. You may add any #includes or constants you like to these files. You may also add support functions for these classes if you like (e.g., *operator*<). Make sure to properly comment your code.

You must submit a brief (You're welcome!) report that presents the big-O for the average case of the following methods. Be sure to make clear the meaning of the variables in your big-O expressions, e.g., "If the MapLoader holds N geo-coordinates, and each geo-coordinate is associated with S geo-segments on average, then *getSegments()* is $O(S^2 \log N)$."

- MyMap: *associate()* and *find()*
- AttractionMapper: *init*(), *getGeoCoord*()
- SegmentMapper: *init*(), *getSegments*()
- Navigator: *navigate()*

Grading

- 90% of your grade will be assigned based on the correctness of your solution
- 5% of your grade will be based on your report
- 5% of your grade will be based on the optimality of your returned routes

Optimality Grading (5%)

Your route-finding algorithm does not need to find an optimal solution to get most of the credit for Project 4; it need only return a valid solution. That said, 5% of the points for Project 4 will be awarded based on the optimality of the paths your algorithm finds.

Given a test case, you will get 1 point for your route for that test case if it is a valid route that is within 10% of our optimal least-total-distance solution for that test case.

We will then total up the number of points you received, divide it by the total number of test cases, and multiply this by 5% to compute your optimality grade (out of a total of 5%).

So if your algorithm were to return an almost-optimal solution (within 10% of our solution) in 35 of our 50 test cases, then you'd get a 70% optimality rating (35/50), and we'd give you 70%*5% points for optimality, or 3.5% out of the possible 5% of your grade for optimality.

Good luck!