Project 4: BruinTour

Time due: 11 PM Thursday, March 14

Introduction

The secretive Bruin Tour Corporation (BTC), the 7th largest business in California, has been responsible for organizing walking tours of UCLA and Westwood for visiting students and parents since UCLA's inception in 1919 and move to Westwood in 1929. In order to save costs by avoiding California's new minimum wage laws, BTC has decided to lay off all of its human tour guides and outsource its tours to robots. Each robot leads a tour of various Westwood landmarks, giving a short talk about each landmark and then navigating to the next landmark.

The BTC leadership team is composed entirely of exceptionally greedy UCLA alumni who decided to "hire" all of our CS 32 students via an unpaid internships to build prototypes of the navigation system for the robots. Lucky you!

So for your last project for CS 32, your goal is to build an automated tour guide system that loads and indexes a bunch of Open Street Map geospatial data (which contain latitude and longitude data for thousands of streets) along with a list of points of interest (aka tour stops like Diddy Riese, Fox Theater, Ackerman Union, etc.) and then uses this data to generate instructions for the new tour robots.

The input to your BruinTour system consists of two files:

1. **mapdata.txt**: A data file that contains details on thousands of streets in Westwood, and lists the latitude and longitude of each street block by block (this data is derived from the Open Street Map service). The data file also contains dozens of "points of interest" (PoIs) that are candidates for the tour (e.g., Diddy Riese, Kerckhoff Hall, etc.)
2. **stops.txt**: A proposed list of points of interest (PoIs) for a given tour along with a description of each PoI for the robot to vocalize, e.g.:

   Ackerman Union|This is Ackerman where you buy stuff.
   John Wooden Center|This is where you go to get swole.
   Diddy Riese|They sell yummy cheap cookies here.

The output of your BruinTour system is a set of instructions for a BruinTour robot to follow in order to give its tour. Instructions fall into three different categories:

- Commentary: Instructs the robot to give commentary about the current PoI, e.g., "John Wooden Center: This is where you get swole."
- Proceed: Instructs the robot to proceed in a particular direction on a street
- Turn: Instructs the robot to make a turn from one street onto another

Given the set of stops shown above, your completed Project 4 solution should be able to produce a set of relatively optimized tour instructions to direct the robot. Our provided main.cpp file will take the tour instructions created by your classes and print something like this:

```
Starting tour...
Welcome to Ackerman Union!
This is Ackerman where you buy stuff.
```

```
Proceed 0.028 miles north on a path
Take a left turn on Bruin Walk
Proceed 0.098 miles west on Bruin Walk
Take a right turn on a path
Proceed 0.074 miles north on a path
Welcome to John Wooden Center!
This is where you go to get swole.
Proceed 0.074 miles south on a path
Take a left turn on Bruin Walk
Proceed 0.043 miles east on Bruin Walk
Take a right turn on Westwood Plaza
Proceed 0.514 miles south on Westwood Plaza
Take a right turn on Le Conte Avenue
Proceed 0.097 miles west on Le Conte Avenue
Take a left turn on Broxton Avenue
Proceed 0.053 miles south on Broxton Avenue
Take a left turn on a path
Proceed 0.015 miles northeast on a path
Welcome to Diddy Riese!
They sell yummy cheap cookies here.
Your tour has finished!
Total tour distance: 0.996 miles
```

If you're able to prove to BTC's reclusive and bizarre co-CEOs, Divad Grebllams and Yerac Grebnehcan, that you have the programming skills to build a simple tour engine, he'll hire you to build the full system, and you'll be rich and famous (at least famous to high school seniors who really want to get into UCLA).

Don't worry – it's easier than it seems!

Anatomy of the BruinTour Engine

To build the BruinTour system, you need to have the following components:

A Geographic Database

All BruinTour systems operate on geolocation data, like the data you can find at the 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 latitude/longitude of various streets and points of interest), 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, but we'll be using data from Open Street Maps because it's freely available. (Just for fun: On a related note, you can get the latitude and longitude for an address at http://www.latlong.net)

The OSM data has geolocation data (latitude, longitude) for each street in its map. 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

You can find these segments starting on line 17011 in the mapdata.txt file we provide.

Notice that for a contiguous stretch of a street, the end latitude/longitude endpoint of one segment is the same as the start lat/long endpoint of another segment, resulting in a contiguous street composed of multiple segments[1]. And here's what Glenmont Ave looks like visually:

[1] Later on, we'll note details about the segments in the mapdata.txt file. For example, although most of the time the segments making up a contiguous stretch of street are in order in the file (so that the end point of one segment is the same start point of the next one in the file, they might not be listed in that order.

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 about 100 feet down and right from Malcolm Ave, at the point at which Glenmont Ave. begins to curve just a 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 in the map above we see intersects Glenmont Ave (in the upper left). Here's a 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 happens to be the point of intersection of Glenmont and Malcolm, and was the first lat/long among the segments we showed you above for Glenmont Ave. We can thus see that these two streets intersect at this point!

So now you understand that **streets are represented by a collection of segments** in the OSM database, where the endpoint of each segment (represented by a latitude/longitude) is connected to 1 or more other endpoints through one or more connecting segments. Of course, all this data is useless unless we organize it so it can be easily searched. That's the purpose of your Geographic Database class. It will hold the coordinates of all the segments as well as information on every point of interest on the map (e.g., the Fox Theater or Diddy Riese).

The Geographic Database class must:

1. Load all of the OSM map data from a text data file into one or more data structures of

your choosing.
2. Given a latitude/longitude coordinate c (we call this a GeoPoint), return all latitude/longitude coordinates (GeoPoints) that are at the other endpoint of street segments that GeoPoint c is one endpoint of, according to the OSM map file.
3. Given a point of interest (e.g., Diddy Riese), return the GeoPoint where that point of interest is located.
4. Given two GeoPoints (which are directly connected by a segment in the OSM map data), return the street name associated with that segment of road (e.g., Westwood Blvd).

Your Routing and Tour Generation classes will make use of your Geographic Database.

A Routing System

A point-to-point routing system leverages the Geographic Database from the section above and a routing algorithm (like A*, or a simple queue-based algorithm) in order to output a series of GeoPoints (latitudes/longitudes) which describe a route from a starting location (as described by a latitude/longitude on the map) to an ending location (as described by another latitude/longitude on the map). Each GeoPoint on the route must be one of the endpoints of the segments held by the Geographic Database, and each GeoPoint is therefore directly connected to the next GeoPoint on the route by a line segment found in the original OSM map data.

For example, here's a route starting from 34.0625329 -118.4470263 (the intersection of Broxton and Weyburn) to 34.0685657 -118.4489289 (the corner of Gayley and Strathmore):

```
34.0625329 -118.4470263 (Intersection of Broxton and Weyburn)
34.0632405 -118.4470467
34.0636533 -118.4470480
34.0636457 -118.4475203
34.0636344 -118.4482275
34.0636214 -118.4491386
34.0640279 -118.4495842
34.0644757 -118.4499587
34.0661337 -118.4484725
34.0664085 -118.4487051
34.0665813 -118.4488486
34.0668106 -118.4490930
34.0670166 -118.4493580
34.0672971 -118.4497256
34.0683569 -118.4491847
34.0684706 -118.4490809
34.0685657 -118.4489289 (Intersection of Gayley and Strathmore)
```

Each GeoPoint above represents a latitude/longitude on the route, and each GeoPoint j is directly connected to the next GeoPoint j+1 via a segment found in the OSM database. For example, you can see that the first two GeoPoints in the route above (34.0625329 -118.4470263 and 34.0632405 -118.4470467) are connected via a segment found at lines 6595-6596 in the mapfile.txt data file, and that this segment is part of Broxton Avenue. Similarly, the second and third GeoPoints (34.0632405 -118.4470467 and 34.0636533 -118.4470480) are connected via a segment found on lines 6592-6593 in the mapdata.txt file.

This list of adjacent GeoPoints could be converted (by another component, discussed in the next section) to a series of human-readable navigation directions through a simple transformation (e.g., measuring the distance and compass direction between each pair of GeoPoints, measuring the angle between segments to determine if a turn onto a new street is a left or right, etc.):

```
Proceed 0.081 miles north on Broxton Avenue
Turn left on Le Conte Avenue
Proceed 0.120 miles west on Le Conte Avenue
Turn right on Levering Avenue
Proceed 0.083 miles northwest on Levering Avenue
Turn right on Roebling Avenue
Proceed 0.146 miles northeast on Roebling Avenue
Turn left on Landfair Avenue
Proceed 0.112 miles northwest on Landfair Avenue
Turn right on Strathmore Drive
Proceed 0.105 miles northeast on Strathmore Drive
```

For this project, you'll need to build a class named *Router* that will leverage your Geographic Database (from above) to compute a series of GeoPoints from a starting latitude/longitude to an ending latitude/longitude that describe a route. The user of this class will request directions from a starting point to an ending point (e.g., from the latitude/longitude of Ackerman Union to the latitude/longitude of Engineering VI).

A Tour Generator

This is where everything comes together. The Tour Generator Component must take as its input a Geographic Database (which knows what GeoPoints connect to other GeoPoints, the names of each street, and the locations of the points of interest) and a set of tour stops and commentary for the robot to say at each stop, and then, using the Routing component, generate a vector of commands that can be interpreted by a robot to deliver a complete tour. Each command in the series of commands outputted by the Tour Generation class will be one of the following:

- **Commentary commands:** instructs the robot to issue commentary about the current point of interest, e.g., "John Wooden Center: This is where you go to get swole"
- **Proceed commands:** instructs the robot to proceed for a particular distance in a particular direction on a street, e.g., "Proceed for 0.302 miles south on Gayley Avenue"
- **Turn commands:** instructs the robot to make a turn from one street onto another, e.g., "Take a left turn on Strathmore Avenue"

So although the Routing component produces a vector of GeoPoints as its result (e.g., 34.0603458 -118.4452615 followed by 34.0597230 -118.4447235, etc.), the Tour Generator component produces a vector of high-level tour commands, where each command is an object that represents a command like proceeding 0.328 miles north on Westwood Blvd or turning right onto Le Conte Avenue, etc.

Our provided main.cpp driver program will use the results of your Tour Generator component to print out a set of human-readable directions:

```
Starting tour...
Welcome to Ackerman Union!
This is Ackerman where you buy stuff.
Proceed 0.028 miles north on a path
Take a left turn on Bruin Walk
Proceed 0.098 miles west on Bruin Walk
Take a right turn on a path
Proceed 0.074 miles north on a path
Welcome to John Wooden Center!
This is where you go to get swole.
Proceed 0.074 miles south on a path
Take a left turn on Bruin Walk
Proceed 0.043 miles east on Bruin Walk
Take a right turn on Westwood Plaza
…
```

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 (whose description is about twice as complicated as a 32-page CS 32 spec). Our *mapdata.txt* file basically has data on thousands of individual street segments, which together make up the entire map. Here's an entry for a particular street called Broxton Avenue from the *mapdata.txt* file:

```
Broxton Avenue
34.0632405 -118.4470467 34.0625329 -118.4470263
3
Diddy Riese|34.0630614 -118.4468781
Fox Theater|34.0626647 -118.4472813
Mr. Noodle|34.0629463 -118.4468728
```

The first line holds the name of the street segment for which we're providing geolocation data. In the example above, we're providing data for one of the segments that make up Broxton Avenue.

The second line holds the starting and ending GeoPoints of this particular street segment in latitude/longitude format as four numbers separated by whitespace: *start_latitude start_longitude end_latitude end_longitude*.

The third line holds a number, call it P, which specifies how many points of interest are directly reachable from this street segment. If P is zero, it means that there are no points of interest to visit on this street segment.

The next P lines specify the name and location of each of the points of interest that may be found along this street segment.

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 GeoPoint, you can determine all GeoPoints directly adjacent to/reachable from that point (i.e., connected by segments listed in the mapdata.txt file). You could then follow each GeoPoint to the GeoPoint at the other end of a segment, and figure out what GeoPoints it's connected to, and so on.

What Do You Need to Do?

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

1. You will create a template class called *HashMap* that works much like the C++ STL *unordered_map* and that implements an open hash table. To ensure that it doesn't exceed a maximum load factor (that is specified as a constructor parameter), a *HashMap* must occasionally grow its number of buckets and rehash its items automatically as you add more data to it. This object will hold associations between a string (e.g., a string representation of a *GeoPoint* or a point of interest name) and an arbitrary type of value (e.g., a vector of GeoPoints that start or end on that *GeoPoint*, or the *GeoPoint* associated with a point of interest name).
2. You will create a class called *GeoDatabase* that will load up all of your map data, store it in efficient data structures, and let you look up GeoPoints and points of interest as described in the sections above.
3. You will create a class called *Router* that allows the user to specify a starting latitude/ longitude location and an ending latitude/longitude location, and which will generate a vector of GeoPoints that describe a path from the starting point to the ending point.
4. You will create a class *TourGenerator* which uses all of the above classes to output a vector of tour commands for a robot that would enable it to deliver a complete tour with commentary and walking directions.

What Will We Provide?

We will provide you with *mapdata.txt*, a large data file that contains mapping data (the names of each street and the latitudes and longitudes for each street segment) for about 20,000 street segments. We've also provided you with tinymapdata.txt which contains a small subset of the data from mapdata.txt for testing more rapidly. During development, you can use this small subset of this file or make up your own small data files for initial testing, and then make sure your program works with the larger file.

We'll also provide you with a number of header files which you may include in your solution but WHICH **YOU MUST NOT MODIFY**:

- *base_classes.h*: Defines a set of base classes named GeoDatabaseBase, RouterBase, and TourGeneratorBase from which you must derive your GeoDatabase, Router, and TourGenerator classes.
- *geopoint.h*: Defines a *GeoPoint* struct that you can use to hold a particular latitude/longitude.
- *geotools.h*: Defines a set of tools to compute the distance in miles, angle, etc. between various GeoPoints.

- *stops.h:* Defines a class which holds all of the stops (aka points of interest) on a tour along with what the tour robot must say about each such stop.
- *tourcmd.h*: Defines a class that represents a tour command that tells the tour robot what to do (e.g., proceed on a street, turn left or right, or comment on the current point of interest).

Finally, we'll provide a simple *main.cpp* file that brings your entire program together and lets you test it.

YOUR CLASSES MUST WORK CORRECTLY with our provided files as is (they MUST NOT be modified to get your solution to work). You will not be submitting these files with your solution.

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 Windows command line, for example, you can run the test harness with:
C:\PATH\TO\YOUR\CODE> BruinTour.exe c:\path\to\your\mapdata.txt c:\path\to\your\stops.txt
You must specify a path to the map data file and a path to a text file that you create which specifies the stops on the tour, and for each stop, a line of commentary to be delivered once that stop is reached. The stops.txt file must have the following format:

Point Of Interest #1 Name|Description of point of interest #1
Point Of Interest #2 Name|Description of point of interest #2
…
Point Of Interest #N Name|Description of point of interest #N

For example, here's an example stops.txt file:

```
Ackerman Union|This is Ackerman where you buy stuff.
John Wooden Center|This is where you go to get swole.
Diddy Riese|They sell yummy cheap cookies here.
Ackerman Union|We're back at Ackerman, and this is the end of your tour.
```

Our main.cpp will take the files you passed on the command line and pass them to your classes so they can load the appropriate map data, generate routes between the various stops, and produce tour commands. Our code in main.cpp will then take the output from your classes and print them to the screen like this:

```
Starting tour...
Welcome to Ackerman Union!
This is Ackerman where you buy stuff.
Proceed 0.028 miles north on a path
Take a left turn on Bruin Walk
Proceed 0.098 miles west on Bruin Walk
Take a right turn on a path
```

```
Proceed 0.074 miles north on a path
Welcome to John Wooden Center!
This is where you go to get swole.
Proceed 0.074 miles south on a path
Take a left turn on Bruin Walk
Proceed 0.043 miles east on Bruin Walk
Take a right turn on Westwood Plaza
Proceed 0.514 miles south on Westwood Plaza
Take a right turn on Le Conte Avenue
Proceed 0.097 miles west on Le Conte Avenue
Take a left turn on Broxton Avenue
Proceed 0.053 miles south on Broxton Avenue
Take a left turn on a path
Proceed 0.015 miles northeast on a path
Welcome to Diddy Riese!
They sell yummy cheap cookies here.
Proceed 0.015 miles southwest on a path
Take a right turn on Broxton Avenue
Proceed 0.053 miles north on Broxton Avenue
Take a right turn on Le Conte Avenue
Proceed 0.097 miles east on Le Conte Avenue
Take a left turn on Westwood Plaza
Proceed 0.514 miles northeast on Westwood Plaza
Take a right turn on Bruin Walk
Proceed 0.055 miles east on Bruin Walk
Take a right turn on a path
Proceed 0.028 miles south on a path
Welcome to Ackerman Union!
We're back at Ackerman, and this is the end of your tour.
Your tour has finished!
Total tour distance: 1.758 miles
```

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/code (**you MUST make no modifications to our provided classes, structs, and functions OR YOU WILL GET A ZERO on this project**).

HashMap

You must implement a class template class named *HashMap* that, like an STL *unordered_map*, lets a client associate strings with related values (e.g., associate "Diddy Riese" with its GeoPoint 34.0630614 -118.4468781). Unlike C++'s unordered_map, your version of the HashMap *always* has a key type of string, so you may only specify the type of the value that's being mapped to.

Your implementation **must** use an open hash table and you may use STL containers EXCEPT for *map*, *unordered_map*, *set*, *unordered_set, multimap, unordered_multimap, multiset,* and *unordered_multiset* to implement your class if you like.

An empty hashmap must start with 10 buckets and have a default maximum load factor of 0.75 if no argument is passed to its constructor.

Here's an example of how you might use *HashMap*:

```
void foo()
{
  // Define a hashmap that maps strings to doubles and has a maximum
  // load factor of 0.3. It will initially have 10 buckets when empty.
  HashMap<double> nameToGPA(0.3);

  // Add new items to the hashmap. Inserting the third item will cause
  // the hashmap to increase the number of buckets (since the maximum
  // load factor is 0.3), forcing a rehash of all items.
  nameToGPA.insert("Carey", 3.5);  // Carey has a 3.5 GPA
  nameToGPA.insert("David", 2.99); // David needs to up his game

  // you can also use brackets like C++'s unordered_map!
  nameToGPA["Annie"] = 3.85; // Adds Annie, who has the highest GPA of all

  double* davidsGPA = nameToGPA.find("David");
  if (davidsGPA != nullptr)
    *davidsGPA = 3.1; // after a re-grade of David's exam, update 2.99 -> 3.1

  nameToGPA.insert("Carey", 4.0); // Carey deserves a 4.0

  // sees if linda is in the map; if not, creates a new entry for linda in map
  cout << nameToGPA["Linda"]; // prints zero
}
```

Your implementation must have the following public interface:

```
template <typename T>
class HashMap
{
public:
  HashMap(double max_load = 0.75); // constructor
  ~HashMap(); // destructor; deletes all of the items in the hashmap
  int size() const; // return the number of associations in the hashmap
  // The insert method associates one item (key) with another (value).
  // If no association currently exists with that key, this method inserts
  // a new association into the hashmap with that key/value pair. If there is
  // already an association with that key in the hashmap, then the item
  // associated with that key is replaced by the second parameter (value).
  // Thus, the hashmap must contain no duplicate keys.
  void insert(const std::string& key, const T& 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 or modify it directly within the map.
  T* find(const std::string& key);
  // Defines the bracket operator for HashMap, so you can use your map like this:
  // your_map["david"] = 2.99;
  // If the key does not exist in the hashmap, this will create a new entry in
  // the hashmap and map it to the default value of type T. Then it will return a
  // reference to the newly created value in the map.
  T& operator[](const std::string& key);
};
```

Requirements for HashMap

Here are the requirements for your HashMap class:

1. You **must** implement your own expandable hash map in your *HashMap* class (i.e., maintain a collection of linked lists of associations, etc.) within hashmap.h.
2. A newly constructed *HashMap* must have 10 buckets and no associations.
3. The *HashMap* constructor parameter is the maximum load factor for the hashmap. If the caller does not pass an argument to the constructor or passes a value that is not positive, then maximum load factor must be 0.75.
4. The key type of a HashMap is always string, but the type of its values must be specified when declaring a variable, e.g., HashMap<int> h; // maps strings to ints.
5. Your *HashMap* class **must** use the public interface shown above. You may add private members to this class, but you **must not** add or change any public members.
6. If a user calls insert() with 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 unique key.
7. If, when adding an item to your expandable hash map, its load factor would increase above the maximum load specified during construction, then you must:
   - Create a new internal hash map with double the current number of buckets.
   - Rehash all items from the current hash map into the new, larger hash map.
   - Replace the current hash map with the new hash map.

• Free the memory associated with the original, smaller hash map.

Note: This means that after a call to any method that might add an association to the table, all pointers into your hash map previously returned by the find() method are potentially invalidated.[2]

8. *HashMap* objects do not need to be copied or assigned. To prevent incorrect copying and assignment of *HashMap* objects, these methods can be declared to be deleted (C++11)[3] or declared private and left unimplemented (pre-C++11).
9. Your member functions **MUST NOT** write anything to *cout*. They may write to *cerr* if you like (to help you with debugging).
10. The Big-O of inserting an item into your hashmap must be O(1) in almost all cases, with the understanding that occasionally an insertion may result in an O(N) insertion time should the hash table need to be resized, where N is the number of associations in the table.
11. The Big-O of searching for an item in your hashmap must be O(1) in the average case, with the understanding that some searches may result in O(N) steps in a pathological case where there are many collisions.
12. The Big-O of getting the number of items in a hashmap with the size() method must be O(1).

You may use the STL facility for hashing strings if you don't want to write your own hash function. It's available by including the header <string>. To hash a string `s`, producing an unsigned integer `h` in the range 0 to at least 4 billion, say `size_t h = std::hash<std::string>()(s);`, noting the two pairs of parentheses.

GeoDatabase Class

The *GeoDatabase* class loads map data from the *mapdata.txt* file we provide and makes the data searchable. The *GeoDatabase* class **must** have the following public interface:

```
class GeoDatabase: public GeoDatabaseBase
{
public:
    GeoDatabase();
    virtual ~GeoDatabase();

    virtual bool load(const std::string& map_data_file);
    virtual bool get_poi_location(const std::string& poi,
                                  GeoPoint& point) const;
    virtual std::vector<GeoPoint> get_connected_points(const GeoPoint& pt)
                                                                  const;
    virtual std::string get_street_name(const GeoPoint& pt1,
                                        const GeoPoint& pt2) const;
};
```

[2] We're not requiring you to do this, but using std::list's *splice* method (or doing something equivalent if you're implementing a list yourself), you can guarantee that the pointers to values in the map will never be invalidated by a rehash because nodes are simply relinked instead of their data being copied elsewhere.

[3] HashMap(const HashMap&) = delete; HashMap& operator=(const HashMap&) = delete;

Here are a few critical requirements for your GeoDatabase class:

1. It MUST be derived from our *GeoDatabaseBase* class, found in base_classes.h.
2. It MUST NOT have any public methods other than those shown above.

GeoDatabase() and possibly ~GeoDatabase()

The constructor must initialize the *GeoDatabase* object. You probably won't need to write much code to do this. This spec imposes no requirements on its time complexity.

Should you need to write one to properly dispose of all memory a *GeoDatabase* object allocates, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

load()

The job of the *load()* method is to load all of the data from an Open Maps data file (e.g., mapdata.txt) and organize it in efficient data structures using your HashMap class to support its operations: get_poi_location, get_connected_points, and get_street_name. Your HashMap MUST NOT use STL containers in this class other than std::vector, std::list, and std::string. Your load() method need not have a particular big-O but should be as efficient as possible. The File I/O and the File input writeups on the main class web page should be helpful.

When loading the street data, you must do the following:

1. For each street segment (that starts on a GeoPoint and ends on a GeoPoint) in the map data file, ensure that the get_connected_points() and get_street_name() can recognize the GeoPoints and produce the correct results.
2. If a street segment S which connects GeoPoints S_1 and S_2 has one or more points of interest on it, then you must do the following:
    a. Determine the midpoint of street segment S using the *midpoi*nt() function found in the provided *geotools.h* file. You **must** use *midpoint()* to determine the midpoint.
    b. Update your internal GeoDatabase data structures to ensure that there is a bidirectional connection between S_1 and the computed midpoint, and assign it a street name that matches the street name of the segment S.
    c. Update your internal GeoDatabase data structures to ensure that there is a bidirectional connection between S_2 and the computed midpoint, and assign it a street name that matches the street name of the segment S.
    d. For each point of interest on that segment:
        i. Update your internal GeoDatabase data structures to ensure that there is a bi-directional connection between the GeoPoint of the point of interest and the computed midpoint GeoPoint which has a street name called "a path".

Why do we do step #2? As a simplifying assumption for this project, we assume that to reach any point of interest (e.g., Diddy Riese) on a street segment, a person would walk along the sidewalk to the midpoint of that street segment, and then follow a straight path between the

midpoint and the point of interest. To get back to the street, the person would walk in a straight line from the point of interest back to the midpoint. From the midpoint, they could then walk to either end of the street segment, which connects with other street segments. If there are 2 or more points of interest on the same street segment, they are not directly connected to each other, but all are connected bidirectionally to the midpoint of that street segment. In this spec, each such path is treated as if it were a street segment for a street named "a path".

This method will open our data file and read the data line by line from this file into your StreetMap (see the File I/O and File input writeups on the class web site). The load() method must return true if the data was loaded successfully, and false otherwise (e.g., if the file could not be found). 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.

get_poi_location()

This method takes in the name of a point of interest, such as Ackerman Union, and then returns the GeoPoint location of that PoI. It returns true if the PoI was found, and sets its second parameter to the GeoPoint of the PoI. Otherwise, it must return false and make no changes to the second GeoPoint parameter. This function must run in O(1) (i.e., constant) time. Here's an example of its usage:

```
int main() {
  GeoDatabase g;
  g.load("mapdata.txt");  // assume this works to avoid error checking
  GeoPoint p;
  if (g.get_poi_location("Diddy Riese", p))
     cout << "The PoI is at " << p.sLatitude << ", "
          << p.sLongitude << endl;
  else
     cout << "PoI not found!\n";
}
```

get_connected_points()

This method takes in a GeoPoint and returns a vector of all GeoPoints that are directly connected to that point via street segment(s). If the function finds no points connected to the searched-for point, then the function returns an empty vector. This function must run in O(p) time where p is the number of GeoPoints that are directly connected to the searched-for point. Here's an example of its usage:

```
int main() {
  GeoDatabase g;
  g.load("mapdata.txt");  // assume this works to avoid error checking
```

```
  std::vector<GeoPoint> pts = g.get_connected_points(
                                    GeoPoint("34.0731003", "-118.4931016"));
  if (pts.empty())
    cout << "There are no points connected to your specified point\n";
  else {
    for (const auto p: pts)
      cout << p.sLatitude << ", "  << p.sLongitude << endl;
  }
}
```

Consider these three street segments taken from the mapdata.txt file:

```
Glenmere Way
34.0732851 -118.4931016 34.0736122 -118.4927669
0
...
Teakwood Road
34.0731714 -118.4921918 34.0736122 -118.4927669
0
Teakwood Road
34.0736122 -118.4927669 34.0739890 -118.4931789
0
```

These describe an intersection where Glenmere and Teakwood meet (at 34.0736122 -118.4927669). If we call get_connected_points() on the point 34.0736122 -118.4927669, the contents of the returned vector should be (in any order):

34.0732851 -118.4931016
34.0731714 -118.4921918
34.0739890 -118.4931789

And, for the following street segment which has points of interest on it:

```
Kinross Avenue
34.0602175 -118.4464952 34.0599361 -118.4469479
2
Fatburger|34.0601422 -118.4468929
Kinross Building North|34.0591552 -118.4463759
```

If we call get_connected_points() on the point 34.0601422, -118.4468929 (the location of Fatburger), the contents of the pts vector should be:

34.0600768, -118.4467216

which is the midpoint of this segment of Kinross Avenue (evenly placed between `34.0602175, -118.4464952` and `34.0599361, -118.4469479`). The midpoint of a street segment **must** be computed by our provided midpoint() function in geotools.h.

Similarly, if we were to call get_connected_points() on the midpoint 34.0600768, -118.4467216, the contents of the pts vector should be (in any order):

```
34.0602175 -118.4464952
34.0599361 -118.4469479
34.0601422 -118.4468929
34.0591552 -118.4463759
```

Where the first two GeoPoints represent the ends of the Kinross Avenue segment, the third GeoPoint represents the Fatburger point of interest, and the fourth GeoPoint would represent the Kinross Building North point of interest.

get_street_name()

A call to *get_street_name()* returns the name of the street segment connected by the two GeoPoints. If the two GeoPoints are not directly connected, then the function must return the empty string. This function must run in O(1) time. It must support the following cases:

1. If the two endpoints queried are the ends of a street segment from the map data file, it must return the street name associated with that segment. Note: The ordering of the two GeoPoints might be different than those in the map data text file. For example, given this street segment:

   ```
   Glenmere Way
   34.0732851 -118.4931016 34.0736122 -118.4927669
   0
   ```

   both of the following calls should set name to "Glenmere Way":

   ```
   GeoPoint p1("34.0732851", "-118.4931016");
   GeoPoint p2("34.0736122", "-118.4927669");
   cout << get_street_name(p1, p2);  // writes "Glenmere Way"
   cout << get_street_name(p2, p1);  // writes "Glenmere Way"
   ```

2. If the two endpoints queried connect a point of interest with a midpoint (or visa-versa) then it must set name to "a path". For example, given this street segment which has a midpoint of 34.0600768, -118.4467216:

```
Kinross Avenue
34.0602175 -118.4464952 34.0599361 -118.4469479
2
Fatburger|34.0601422 -118.4468929
Kinross Building North|34.0591552 -118.4463759
```

both of the following calls should set name to "a path":

```
GeoPoint p1("34.0601422", "-118.4468929");
GeoPoint p2("34.0600768", "-118.4467216");
cout << get_street_name(p1, p2);  // writes "a path"
cout << get_street_name(p2, p1);  // writes "a path"
```

3. If the two endpoints queried connect a midpoint with the end of a street segment (or visa-versa) then your method must return the street name associated with the overall segment that the midpoint is located on. For example, given this street segment which has a midpoint of 34.0600768, -118.4467216:

```
Kinross Avenue
34.0602175 -118.4464952 34.0599361 -118.4469479
2
Fatburger|34.0601422 -118.4468929
Kinross Building North|34.0591552 -118.4463759
```

both of the following calls should set name to "Kinross Avenue":

```
GeoPoint p1("34.0602175", "-118.4464952");
GeoPoint p2("34.0600768", "-118.4467216");
cout << get_street_name(p1, p2);  // writes "Kinross Avenue"
cout << get_street_name(p2, p1);  // writes "Kinross Avenue"
```

Router Class

The *Router* class is responsible for computing an efficient route from a source GeoPoint s to a destination GeoPoint d, if one exists (a "route" is a vector of GeoPoints that are directly connected to each other, including points s and d). It must use the *GeoDatabase* class to stitch together a set of waypoints (GeoPoints) between the starting and ending GeoPoints.

```
class Router: public RouterBase
{
public:
   Router(const GeoDatabaseBase& geo_db);
   virtual ~Router();
   virtual std::vector<GeoPoint> route(const GeoPoint& pt1,
                                       const GeoPoint& pt2) const;
};
```

Here are critical requirements for your Router class:

1. It MUST be derived from our RouterBase class, found in base_classes.h.
2. It MUST NOT have any public methods other than those shown above.
3. The Router class may use any STL containers you like (e.g., map, set, vector, list, queue, etc.)
4. It must not directly access any other classes that you write (e.g., GeoDatabase) but instead must use their Base classes (e.g., GeoDatabaseBase). The one exception is that your Router class may use the HashMap class that you've written.
5. It must **not** write anything to *cout*. It may write to *cerr* if you wish (to help you with debugging).
6. Assuming there are N GeoPoints in our mapping data, your *route*() method must run in O(N*logN) time or less. However, if you implement your route-finding algorithm efficiently, it should generally run in far less time.

Router() and possibly ~Router()

Your *Router* constructor must accept a reference to a fully-constructed *GeoDatabase* object containing loaded street map data. Notice that this constructor doesn't take a reference to a *GeoDatabase* object as a parameter, but to a *GeoDatabaseBase* object. This way, if you have bugs in your *GeoDatabase* class, we can substitute our correct version of the class for yours and then test your *Router* class with our correct version. This will help you get points for one class even if another class it relies upon has bugs. There is no Big-O requirement for your Router constructor.

Should you need to write a destructor to properly dispose of all memory a *Router* object allocates, you must declare and implement a destructor that does so. This spec imposes no requirements on its Big-O.

route()

After constructing a *Router* object, its user may call *route()* to compute a "path" - a vector of GeoPoints that lead from a *starting GeoPoint (pt1)* to an ending *GeoPoint (pt2)*. The returned *path* may contain both GeoPoints explicitly found in your mapdata.text file (e.g., the ends of street segments or the locations of points of interest) as well as GeoPoints corresponding to the segment midpoints that implicitly connect with the points of interest along that segment (and the segment's end GeoPoints). Every GeoPoint in the path generated by your route() method must

be directly connected with the previous and following GeoPoints, creating a contiguous path through the street segments that make up the map. Assuming a valid set of connecting *GeoPoints* can be found between the two coordinates, the vector this method returns must have a starting *GeoPoint* that matches the passed-in `pt1` parameter, and the last *GeoPoint* in the vector must match the `pt2` parameter. If no path was found, the returned vector must be empty. If the `pt1` and `pt2` parameters are the same (e.g., you're navigating to where you already are standing), then the vector your method returns must have just one element equal to that *GeoPoint*: A route from a position to itself requires no travel.

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

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… ". But in fact, it's possible to implement an optimal routing algorithm in just a few hundred lines of code (or less!) using a simple STL *queue* and STL *set* or even better, 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 *GeoDatabase* class to locate adjacent 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 list of GeoPoints to provide point-to-point 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 (point-by-point directions) back to the user. In your maze searching homework, you simply returned *true* or *false* indicating whether a path through the maze existed.., you didn't return the actual steps to get through the maze! To get credit for the *Router* class, you have to return each GeoPoint that makes up that route!

How might you track the complete point-by-point 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 *HashMap* class) that associates a given GeoPoint G to the previous GeoPoint P in the route (e.g., we traveled **to** G directly **from** P). Let's call this map: *locationOfPreviousWayPoint*

Let's illustrate the approach with an example containing 4 GeoPoints: A, B, C and D where A and B are connected, B and C are connected, and C and D are connected.

Let's assume that your queue-based search algorithm is searching for a path from A to D. The algorithm could look up point A using the *GeoDatabase* class, and find the connected point B. At this point, the algorithm would queue GeoPoint B to be processed later, and add the following association to your map:

locationOfPreviousWayPoint[B] -> A

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

Next, the algorithm might dequeue GeoPoint B, and then use the *GeoDatabase* class to determine that it can reach GeoPoint 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 the algorithm would dequeue GeoPoint C from the queue. Again, using the *GeoDatabase* class, it would determine that point C leads to point D. Again, it could add this fact to its 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 GeoPoint that we traveled *from* to get to that waypoint.

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

locationOfPreviousWayPoint[B] -> 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 look up 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 GeoPoints can be added to an STL vector and then returned to the user.

Since, like the maze searching algorithm, your routing algorithm must only visit each GeoPoint 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 GeoPoint, 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 coordinate in concentric growing rings. It starts by locating all *GeoPoints* connected to our starting coordinate and adds them to our queue. It then finds all GeoPoints connected with these *GeoPoints* and adds the other end coordinates to our queue. And so on, and so on. Eventually, either the algorithm stumbles upon the ending coordinate, or it will work its 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 coordinate, 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 coordinate.

Now if you think hard, you'll realize that this algorithm won't necessarily find the shortest path (in miles) from your source coordinate to the destination coordinate. 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 smallest 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 directly connects A to C (100 miles) via a single segment, 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 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 GeoPoint X. Further, let's assume that X is connected to three outgoing points Y, Z and 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 coordinate. 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.

TourGenerator Class

The *TourGenerator* class is responsible for producing tour instructions for a robot to follow given an input set of points of interest that need to be visited on the tour. This class brings all of your other classes together to solve the overall problem of BruinTour.

```
class TourGenerator: public TourGeneratorBase
{
public:
    TourGenerator(const GeoDatabaseBase& geodb, const RouterBase& router);
    virtual ~TourGenerator();
    virtual std::vector<TourCommand> generate_tour(const Stops &stops) const;
};
```

Requirements for *TourGenerator*

Here are the requirements for your *TourGenerator* class:

1. It MUST be derived from our TourGeneratorBase class, found in base_classes.h.
2. It MUST NOT have any public methods other than those shown above.
3. The TourGenerator class may use any STL containers you like (e.g., map, set, vector, list, queue, etc.)
4. It must not directly access any other classes that you write (e.g., GeoDatabase, Router) but instead must use their Base classes (e.g., GeoDatabaseBase, RouterBase). The one exception is that your TourGenerator class may use the HashMap class that you've written.
5. It must **not** write anything to *cout*. It may write to *cerr* if you wish (to help you with debugging).
6. Assuming there are N GeoPoints in our mapping data, and there are P points of interest on the tour, your generate_tour method must run in O(P*N*logN) time or less. However, if you implement your tour generator efficiently, it should generally run in far less time.

TourGenerator() and possibly ~TourGenerator()

Your constructor takes a reference to a *GeoDatabaseBase* object and a reference to a RouterBase object. Our main.cpp code instantiates your GeoDatabase and Router classes and passes them into your TourGenerator constructor for you. There are no requirements on the constructor's Big-O.

Should you need to write one to properly dispose of all memory a *TourGenerator* object allocates, you must declare and implement a destructor that does so. There are no requirements on its time complexity.

generate_tour()

This method must take in a *Stops* object (which holds the names of all the points of interest on the tour as well as a description of each PoI) and returns a vector of *TourCommand* objects that it has filled with *TourCommands* (see *tourcmd.h*) to direct a tour robot. The method generates a series of TourCommands that would direct a robot to all the stops on the tour in the order they occur in the Stops object. A *TourCommand* represents a command that would be given to a tour robot to instruct it what to do at a given step of a tour:

- **Give commentary on a PoI**: Output a description of the tour stop to the guests
- **Proceed on a street**: Proceed on a street segment of a particular name for a particular distance (in miles) in a particular direction
- **Turn on a street**: Turn left or right from the current street onto a street of a given name

If no route is possible or an unknown point of interest is passed in as a stop, then your method must return an empty vector.

To generate these *TourCommands*, you may use something like the following pseudocode:

- Create a result vector that will hold the instructions
- For each point of interest P held by the Stops object:
    - Generate a **commentary** TourCommand object that specifies what should be said about the current point of interest P and add it to the end of the instructions result vector
    - If there is another point of interest following P then use your Router class to generate a point-to-point route between the GeoPoint associated with P and the GeoPoint of the next point of interest, and then…
    - Using the path of GeoPoints generated by *Router* object in the previous step*, g*enerate a sequence of *TourCommands* representing **proceed** and **turn** instructions for the tour robot:
        - You must add a **proceed** tour command to the output vector for every two adjacent GeoPoints p1 and p2 on the path.
        - You must add a **turn** tour command to the output vector after each **proceed** tour command if and only if:
            - there is a GeoPoint p3 directly after p2 on the path, and…
            - the street/path name of the first segment (p1 → p2) and the street/path name of the second segment (p2 → p3) differ, and…
            - the angle between the two mathematical vectors created by p1 → p2, and p2 → p3 is not equal to zero (i.e., there is some sort of turn); if the vectors p1 → p2 and p2 → p3 are in the exact same direction, then a **turn** command must not be added.

Upon completion, a BruinTour robot should be able to use the tour instructions produced to give a full tour from start to finish, in the order specified by the Stops object.

The sequence of tour commands produced must meet the following requirements:

- Every **commentary** TourCommand must have the name of the point of interest and the talking points
- Every **proceed** TourCommand represents movement from a first GeoPoint p1 to a connected GeoPoint p2, and must have a street name (which would be the name "a path" for a movement to/from a point of interest), a distance (in miles), and the direction of travel (e.g., southwest, north) based on the angle of direction of the mathematical vector pointing from from p1 → p2:
    - 0 <= angle < 22.5: east
    - 22.5 <= angle < 67.5: northeast
    - 67.5 <= angle < 112.5: north
    - 112.5 <= angle < 157.5: northwest
    - 157.5 <= angle < 202.5: west
    - 202.5 <= angle < 247.5: southwest
    - 247.5 <= angle < 292.5: south
    - 292.5 <= angle < 337.5: southeast
    - angle >= 337.5: east
- Every **turn** TourCommand must have the direction to turn, ("left" or "right") and the name of the street/path to turn onto (e.g., "Levering Avenue"):
    - >= 1 degrees and < 180 degrees: Generate a ***left*** turn command.
    - >= 180 degrees and <= 359 degrees: Generate a ***right*** turn command.
- Never generate a ***turn*** TourCommand as your first command when leaving a point of interest – assume that when you leave the point of interest you are always facing in the direction you need to go on the first street you intend to travel on, and therefore don't need turn left or right to start traveling on it. Therefore, always generate a ***proceed*** command as the first *TourCommand* after a **commentary** TourCommand

You will likely find the functions we provide you in *geotools.h* very useful for directional and distance computations.

To illustrate the above with an example, let's say that the following stops are provided to your *TourGenerator* class via the following *stops.txt* file, which can be loaded by the Stops class we provide:

```
Diddy Riese|This is where you get cheap yummy cookies.
Ami Sushi|Enjoy some raw fish here.
Fox Theater|Watch a movie premiere here with the big shots.
```

Furthermore, here are relevant excerpts from your mapdata.txt data file:

```
Broxton Avenue
34.0632405 -118.4470467 34.0625329 -118.4470263
3
Diddy Riese|34.0630614 -118.4468781
Fox Theater|34.0626647 -118.4472813
Mr. Noodle|34.0629463 -118.4468728
Broxton Avenue
34.0625329 -118.4470263 34.0624128 -118.4470197
0
```

```
Broxton Avenue
34.0624128 -118.4470197 34.0620596 -118.4467237
0
Broxton Avenue
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
```

In the above example, your *Router* class might output the following *GeoPoints* from *Diddy Riese* to *Ami Sushi*:

34.0630614, -118.4468781 // Diddy Riese location
34.0628867, -118.4470365 // Midpoint of Broxton Ave where Diddy Riese is
34.0625329, -118.4470263 // End of Broxton segment
34.0624128, -118.4470197 // End of Broxton segment
34.0620596, -118.4467237 // End of Broxton segment
34.0616960, -118.4464189 // Midpoint of Broxton segment where Ami Sushi is
34.0614911, -118.4464410 // Ami Sushi location

And your Router class might output the following GeoPoints from Ami Sushi to the Fox Theater:

34.0614911, -118.4464410 // Ami Sushi location
34.0616960, -118.4464189 // Midpoint of Broxton segment where Ami Sushi is
34.0620596, -118.4467237 // End of Broxton segment
34.0624128, -118.4470197 // End of Broxton segment
34.0625329, -118.4470263 // End of Broxton segment
34.0628867, -118.4470365 // Midpoint of Broxton Ave where Fox Theater is
34.0626647, -118.4472813 // Fox Theater location

These would be converted into the following TourCommands:

Commentary: Diddy Riese - This is where you get cheap yummy cookies.
Proceed: 0.0150964 miles southwest on a path, from 34.0630614 -118.4468781 to 34.0628867 -118.4470365
Turn: left onto Broxton Avenue
Proceed: 0.0244522 miles south on Broxton Avenue, from 34.0628867 -118.4470365 to 34.0625329 -118.4470263
Proceed: 0.0083067 miles south on Broxton Avenue, from 34.0625329 -118.4470263 to 34.0624128 -118.4470197
Proceed: 0.0297086 miles southeast on Broxton Avenue, from 34.0624128 -118.4470197 to 34.0620596 -118.4467237

Proceed: 0.0305906 miles southeast on Broxton Avenue, from 34.0620596 -118.4467237 to 34.0616960 -118.4464189
Turn: right onto a path
Proceed: 0.0142104 miles south on a path, from 34.0616960 -118.4464189 to 34.0614911 -118.4464410
Commentary: Ami Sushi - Enjoy some raw fish here.
Proceed: 0.0142104 miles north on a path, from 34.0614911 -118.4464410 to 34.061696 -118.4464189
Turn: left onto Broxton Avenue
Proceed: 0.0305906 miles northwest on Broxton Avenue, from 34.0616960 -118.4464189 to 34.0620596 -118.4467237
Proceed: 0.0297086 miles northwest on Broxton Avenue, from 34.0620596 -118.4467237 to 34.0624128 -118.4470197
Proceed: 0.0083067 miles north on Broxton Avenue, from 34.0624128 -118.4470197 to 34.0625329 -118.4470263
Proceed: 0.0244522 miles north on Broxton Avenue, from 34.0625329 -118.4470263 to 34.0628867 -118.4470365
Turn: left onto a path
Proceed: 0.0207753 miles southwest on a path, from 34.0628867 -118.4470365 to 34.0626647 -118.4472813
Commentary: Fox Theater - watch a movie premiere here with the big shots.

Project Requirements and Other Thoughts

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

1. In Visual C++, change your project from UNICODE to Multi Byte Character set, by going to Project / Properties / Configuration Properties / Advanced (or General) / Character Set. (This might not be necessary in Visual Studio 2022.)
2. The entire project can be completed in under 600 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 classes (e.g., *Router*, *GeoDatabase, TourGenerator*, etc.) must **never** ***directly*** refer to your other classes. They MUST refer to our provided base classes instead:

FORBIDDEN FOR THIS PROJECT:

```
class TourGenerator
{
  ...
private:
  Router& m_router; // NO!
  ...
};
```

PERMITTED:

```
class TourGenerator
{
 ...
private:
  RouterBase& m_router; // OK!
 ...
};
```

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 explicitly permitted by this spec. Use the *HashMap* class in your *GeoDatabase* implementation if you need a map, for example; do **not** use the STL *map* or *unordered_map* class.
9. You may use STL algorithms, such as *sort()*. If you do, make sure to include <algorithm>.
10. Let *Whatever* represent *GeoDatabase*, *Router*, and *TourGenerator*. Subject to the constraints we imposed (e.g., no changes to the public interface of the *Whatever* class, no mention of *Whatever* 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 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 *HashMap* class working with your own hash table, create a simple version of your HashMap class implemented using the STL unordered_map, and go back to fixing your *HashMap* 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., *GeoDatabase*), we will provide a correct version of that class and test it with the rest of your program (by building with our *GeoDatabase.h/cpp* that uses a correct, finished 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 *GeoDatabase* class and we can give you credit for those parts of the project you completed (This is why we're using *derived* classes and base 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 **eight to ten** files. Eight are required and two are optional:

HashMap.h Contains your hash map class template implementation
geodb.h/geodb.cpp Contains your street map index
router.h/ router.cpp Contains your point to point router
tour_generator.h/ tour_generator.cpp Contains your overall tour planner
report.txt or report.docx Contains your report
support.h/support.cpp You may define support constants/classes/functions in these support 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 *GeoDatabase* holds N GeoPoints, and each GeoPoint is associated with P other GeoPoints on average, *get_connected_points()* is $O(P^2 \log N)$." Give the Big-O for these methods in your report:

- GeoDatabase: *load*(), *get_connected_points*(), get_street_name()
- Router: *route*()

Grading

- 90% of your grade will be assigned based on the correctness of your solution
- 5% of your grade will be based on the optimality of your tour routes (shorter is better)
- 5% of your grade will be based on your report

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 routes your algorithm finds.

We'll run several tests for optimality. You'll earn credit for an optimality test if your route for that test case if it is a valid route that is shorter than 110% of our optimal least-total-distance solution for that test case and takes less than 10 times as long to compute as ours or 3 seconds, whichever is longer.

Good luck!