Programming Assignment #4

Due: 2015/11/12 at 11:59pm
Points: 20

Goal

To combine various data structures we've learned to implement geocoding.

Background

We're continuing on the theme of maps. In PG5, you will write a program to compute driving directions between two points in South Bend. In this programming assignment, you will implement *geocoding*, which means converting a human-readable address like "1600 Pennsylvania Ave" into a machine-readable format. Usually the output is a latitude and longitude, but for this assignment, we won't need to pinpoint the location so precisely; instead, we'll find the street segment that the address lies on.

Distribution

Update your local Git repository (as described at `https://bitbucket.org/CSE-30331-FA15/cse-30331-fa15`) and change to the directory `PG4`. It contains the following files:

`pg4.pdf`	this file
`southbend.map`	a map of South Bend and Notre Dame
`Makefile`	simple Makefile
`test{1,2}.cpp`	test programs
`test.sh`	test script
`README.md`	README

The first few lines of `southbend.map` are:

```
N: Bercliff Dr
R: 0 300 314 11397628 11397633 0.0361305195783 https://maps.googleapis.com/...
R: 0 316 332 11397633 11397632 0.0482335773516 https://maps.googleapis.com/...
R: 0 334 398 11397632 11397627 0.0391453933473 https://maps.googleapis.com/...
R: 1 301 309 11397628 11397633 0.0361305195783 https://maps.googleapis.com/...
R: 1 311 331 11397633 11397632 0.0482335773516 https://maps.googleapis.com/...
R: 1 333 399 11397632 11397627 0.0391453933473 https://maps.googleapis.com/...
N: Echo Dr
R: 0 1100 1174 11395956 11401670 0.093521317483 https:...
```

Each line begins with either `N:` or `R:`. A line that starts with `N:` gives the name of a street. The subsequent lines starting with `R:` each describe a range of odd or even addresses on that street, using seven whitespace-separate fields:

1. A *parity*: `0` means even, `1` means odd.
2. The starting address of the range (inclusive).

3. The ending address of the range (inclusive).
4. The ID of the starting *node*. Ignore this field for PG4.
5. The ID of the ending *node*. Ignore this field for PG4.
6. The length in miles of this street segment. Ignore this field for PG4.
7. The URL of an image showing this street segment on a map.

Thus, the second line means that even addresses on Bercliff Dr between 300 and 314 inclusive lie on a 0.036-mile street segment running from node 11397628 to node 11397633, which you can display by visiting the URL `http://maps.googleapis.com/...` (the full URL is not shown here).

You may assume that if two segments are on the same street with the same parity, they do not overlap.

Task

This assignment is a little different from previous assignments because it focuses on making data structures and algorithms work together (viewing them from the outside) rather than their internals. You can use any class or function from the standard library you want. You're encouraged not to use any pointers (`new`/`delete`) or arrays at all; indeed, this is the norm for most idiomatic C++ code that you will write.

You will write a class called `street_map` that has two methods:

```
class street_map {
public:
  explicit street_map(const std::string &filename);
  bool geocode(const std::string &address, std::string &url);
};
```

The constructor should take a filename (`"southbend.map"`) as an argument and read all the data from that file.

The function `geocode` should take an address (e.g., `"1417 E Wayne St"`) and return `true` if the address was found and `false` otherwise; if it was found, it should put the URL of the containing street segment into argument `url`. It will work in several steps:

- Parse the address into a house number and a street name.
- Look up the street segments for the even or odd side of the street.
- Within those street segments, find the segment that includes the house number.
- Print out the URL of the street segment.

1 Write a class (or struct) called `side` to represent a *side* of a street (that is, the odd-numbered side or the even-numbered side). It should have members for the street name and for the parity.

2 Write a class (or struct) called `segment` to represent a street segment. It should have members for starting and ending house number, and URL.

3 The files given already have a skeleton class called `street_map`. Design an appropriate data structure for holding all the data and make it a (private) member of `street_map`. It must have a type of the form

```
std::unordered_map<side, ...>
```

where you have to fill in the `...`. (What two functions do you need to define in order to create an `unordered_map` with `side` as its keys?)

4 Write code to read `southbend.map` into your data structure.

5 Write code to parse an address into a house number and a street name, and to look up the street segments for the appropriate side of the street.

6 Write code to search for the street segment that contains the address. (See the next step for some hints.) At this point, `test1` should pass all tests.

7 Try running `./measure ./test2`. Does it pass the speed and memory tests (see rubric below)? Depending on your choice of data structures, you might find it helpful to use `std::lower_bound`, which uses binary search and is guaranteed to run in logarithmic time. It expects the thing you're searching for and the things you're searching among to have the same type. So if you're searching for a single house number *a* in a sequence of street segments, you can make a dummy street segment with starting and ending house number *a* and empty URL.

8 In `README.md`, write a short description of the data structures and algorithms you used, and any other feedback you'd like to offer about what you learned or didn't learn from this assignment.

Rubric

`test1` and `test2` compile	1
class `side` looks correct	1
class `segment` looks correct	1
`street_map` members look correct	1
`street_map` constructor looks correct	3
`street_map::geocode` looks correct	3
`test1` passes correctness tests	3
`test1` passes Valgrind	1
`test2` runs in 5 seconds or less	3
Good style	1
Good comments	1
Good README	1
Total:	20

Submission

1. To submit your work, upload it to your repository on Bitbucket:

   (a) `git add` *`filename`* for every file that you created or modified

   (b) `git commit -m` *`message`*
   (c) `git push`

   The last commit made at or before 11:59pm will be the one graded.

2. Double-check your submission by cloning your repository to a new directory, running `make` and running all tests.