CS 211
Fundamentals of Computer Programming II
Winter 2024

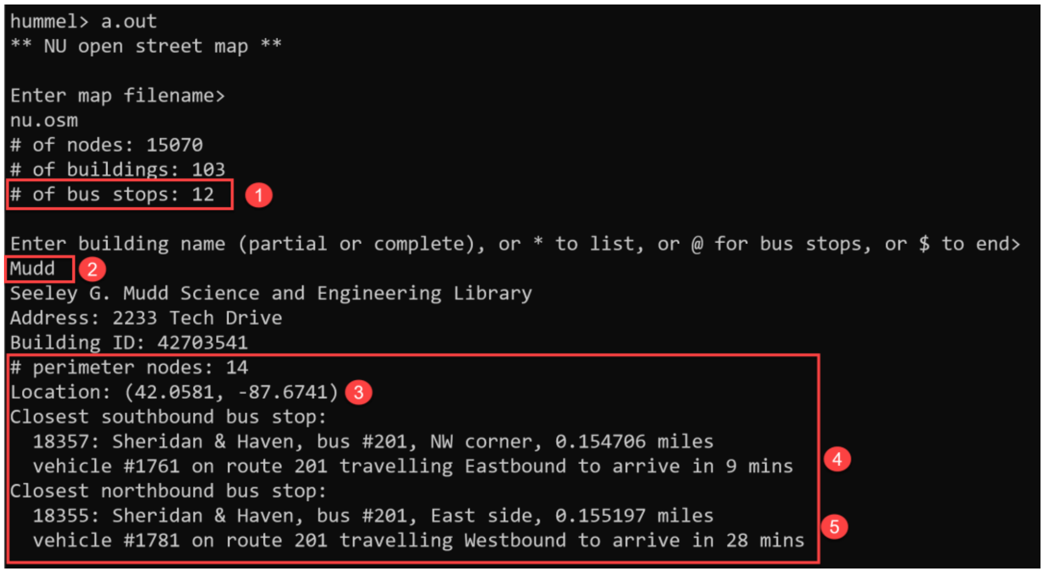
Project #07 (v1.2)

Assignment: Open Street Maps --- nearest bus stops
Submission: via Gradescope (4 submissions per 24-hr period)
Policy: individual work only, late work is accepted
Complete By: Friday March 1st @ 11:59pm CST
Late submissions: see syllabus for late policy… No submissions accepted after Sunday 03/03 @ 11:59pm

Pre-requisites: HW #07 (reading CSV data and sorting objects)

Overview

In projects 05 and 06 we worked with an open street map of NU's Evanston campus. Here in project 07 we're going integrate CTA bus stops into the map, so that when the user enters a building name, the program can output the nearest southbound and northbound stops. Your program will also query the CTA's online bus tracker API to get a prediction on when the next bus(es) may be arriving:



```
hummel> a.out
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of bus stops: 12

Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
Mudd
Seeley G. Mudd Science and Engineering Library
Address: 2233 Tech Drive
Building ID: 42703541
# perimeter nodes: 14
Location: (42.0581, -87.6741)
Closest southbound bus stop:
  18357: Sheridan & Haven, bus #201, NW corner, 0.154706 miles
  vehicle #1761 on route 201 travelling Eastbound to arrive in 9 mins
Closest northbound bus stop:
  18355: Sheridan & Haven, bus #201, East side, 0.155197 miles
  vehicle #1781 on route 201 travelling Westbound to arrive in 28 mins
```

Step #1 shows that 12 CTA bus stops were loaded when the program started. Buildings now have a location computed from their perimeter nodes (step #3), which is used to compute the closest CTA bus stops. Predictions are then downloaded via the CTA's online bus tracker API for these stops (steps #4 and #5).

Getting Started

We will continue working on the EECS computers. The first step is to setup your project07 folder and copy the necessary files (these files can also be found on dropbox):

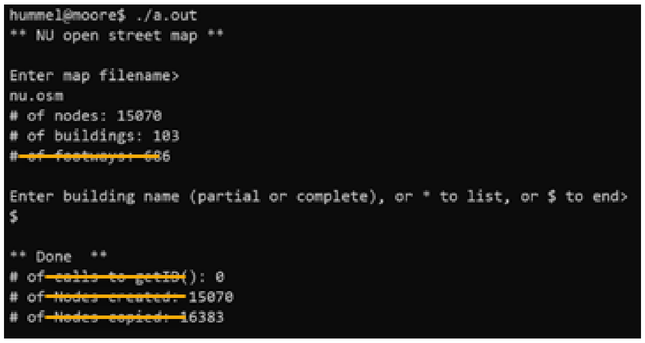
1. Open a terminal window in VS Code. Or, if you are working on a Mac or Linux, open a terminal; if you are working on Windows, you can open Git Bash.
2. Login to moore: **`ssh moore`** or **`ssh YOUR_NETID@moore.wot.eecs.northwestern.edu`**
3. Make a directory for project 07 **`mkdir project07`**
4. Make this directory private **`chmod 700 project07`**
5. Move ("change") into this directory **`cd project07`**
6. Copy the files needed for project 07:
    a. To use <u>your</u> solution:
    ```
    cp -r ../project06/release .
    cd release
    cp /home/cs211/w2024/project07/util/* .
    ```
    b. To use <u>our</u> solution:
    ```
    cp -r /home/cs211/w2024/project07/release .
    cd release
    ```
7. List the contents of the directory **`ls`**

<u>If you are working from your own solution</u>, you will need to modify your makefile to (a) compile the provided C++ files "dist.cpp" and "curl_util.cpp", and (b) link in the CURL library used for internet communications. The necessary changes are shown below in **boldface**:

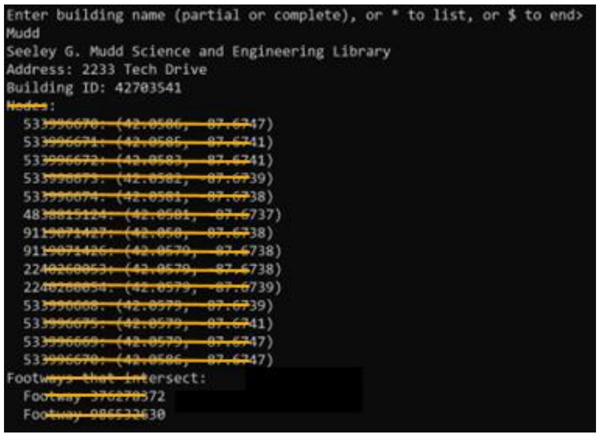
```
build:
  rm -f ./a.out
  g++ -std=c++17 -g -Wall <<your .cpp files>> dist.cpp curl_util.cpp -Wno-unused-
variable -Wno-unused-function -lcurl
```

Similar changes should be made to any other g++ command in the makefile (e.g. valgrind?).

At this point you should be able to build the program (**make build**) without error. Now modify the program to eliminate some of the output from projects 05 and 06 if present: (a) do not print the # of footways, (b) do not print the stats at the end, (c) do not print a building's nodes, and (d) do not print the intersecting footways:



```
hummel@moore$ ./a.out
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
#~~ of footways: 6~~86

Enter building name (partial or complete), or * to list, or $ to end>
$

** Done  **
# of~~ calls to getID~~(): 0
# of~~ Nodes created:~~ 15070
# of~~ Nodes copied:~~ 16383
```



```
Enter building name (partial or complete), or * to list, or $ to end>
Mudd
Seeley G. Mudd Science and Engineering Library
Address: 2233 Tech Drive
Building ID: 42703541
~~Nodes~~:
  53~~3996670: (42.0586, -87.67~~47)
  53~~3996671: (42.0585, -87.67~~41)
  53~~3996672: (42.0583, -87.67~~41)
  53~~3990673: (42.0582, -87.67~~39)
  53~~3990674: (42.0581, -87.67~~38)
  48~~38815124: (42.0581, -87.~~6737)
  91~~19071427: (42.058, -87.67~~38)
  91~~19071426: (42.0579, -87.~~6738)
  22~~40260053: (42.0579, -87.~~6738)
  22~~40260054: (42.0579, -87.~~6739)
  53~~3990668: (42.0579, -87.67~~39)
  53~~3996675: (42.0579, -87.67~~41)
  53~~3996669: (42.0579, -87.67~~47)
  53~~3996670: (42.0586, -87.67~~47)
Foot~~ways that int~~ersect:
  Foo~~tway 3762703~~72
  Foo~~tway 9865326~~30
```
[redacted]

Step 01: getline( )

When inputting the command from the user, be sure you are using **getline(cin, command)** to input the command / building name. Why? Because the building name may contain multiple words such as “University Library”, in which case the >> operator only inputs the first word; the solution is to use getline( ).

Interestingly, when you switch to using getline( ), you have to be consistent in that all keyboard input is obtained via getline( ) --- mixing >> and getline( ) does not work well. This implies that you should also use **getline(cin, filename)** to input the name of the OSM file.

Step 02: building location

In order to find bus stops near a building, every building will need a location --- a position in latitude and longitude. Add a public method to the Building class with the following design:

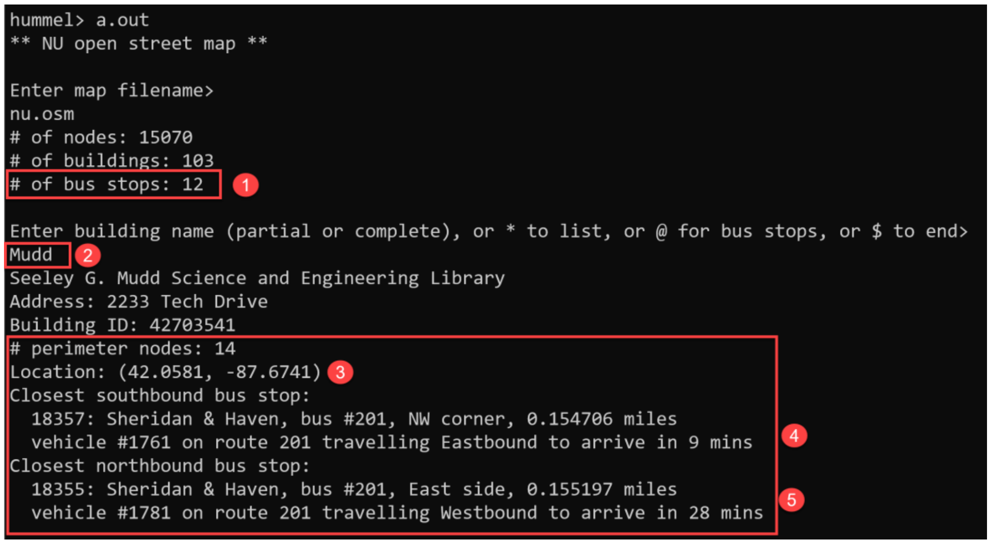
```
//
// gets the center (lat, lon) of the building based
// on the nodes that form the perimeter
//
pair<double, double> getLocation(const Nodes& nodes);
```

The building’s location is the average latitude and longitude of the nodes that form the perimiter. The function returns the latitude and longitude as a pair<double, double>. To return a pair, first *#include <utility>*, and then use the *make_pair( )* function:

```
return make_pair(avgLat, avgLon);
```

NOTE: creating this function is a requirement of the assignment. No nested loops.

When you’re done, modify your Building print( ) function to output the # of perimeter nodes along with the building’s location. For an example, see step #3 in the following screenshot:



```
hummel> a.out
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of bus stops: 12     1

Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
Mudd    2
Seeley G. Mudd Science and Engineering Library
Address: 2233 Tech Drive
Building ID: 42703541
# perimeter nodes: 14
Location: (42.0581, -87.6741) 3
Closest southbound bus stop:
  18357: Sheridan & Haven, bus #201, NW corner, 0.154706 miles
  vehicle #1761 on route 201 travelling Eastbound to arrive in 9 mins    4
Closest northbound bus stop:
  18355: Sheridan & Haven, bus #201, East side, 0.155197 miles
  vehicle #1781 on route 201 travelling Westbound to arrive in 28 mins   5
```

Step 03: CTA bus stops

Information about the CTA bus stops will be input from a CSV file named “bus-stops.txt”. You can open and review this file using VS Code. The provided file contains 12 bus stops, one per line. Each bus stop consists of a stop ID (15924), the bus route (201), the stop name (Central & Sheridan), the direction of travel (Eastbound), the location of the stop (SW corner), and the position in latitude and longitude:

```
15924,201,Central & Sheridan,Eastbound,SW corner,42.06401,-87.67725
17301,201,Sheridan & Lincoln,Southbound,SW corner,42.06144,-87.67730
18356,201,Sheridan & Lincoln,Northbound,NE corner,42.06178,-87.67696
18357,201,Sheridan & Haven,Southbound,NW corner,42.05807,-87.67708
18355,201,Sheridan & Haven,Northbound,East side,42.05756,-87.67699
17295,201,Sheridan & Foster,Southbound,SW corner,42.05388,-87.67737
17296,201,Sheridan & Foster,Northbound,East side,42.05419,-87.67710
18354,201,Sheridan & Emerson,Northbound,East side,42.05228,-87.67712
17302,201,Chicago & Sheridan,Southbound,SW corner,42.05086,-87.67756
15318,201,Chicago & Clark,Southbound,NW corner,42.04969,-87.67816
14403,201,Chicago Ave & Church,Southbound,NW corner,42.04797,-87.67897
14692,201,Chicago Ave & Davis,Southbound,NW corner,42.04643,-87.67970
```

You may assume each line of the file has this format, and no error checking of the data is required. However, make no other assumptions about the file --- do not assume a particular order, do not assume these exact stops, do not assume the # of stops, etc. We reserve the right to change the contents of the file for testing purposes.

Create a **BusStop** class to model one bus stop. Then create a **BusStops** class to store all the bus stops, and provide additional functionality. *In other words, continue the design we have been following: Building / Buildings, Node / Nodes, etc.* When creating your .h and .cpp files, use all lowercase --- name the files “busstop.h”, “busstop.cpp”, “busstops.h” and “busstops.cpp” (this is the convention in C/C++). The constructor for BusStops should take the filename of the CSV file, input the data and build a container of BusStop objects; recall that HW #07 showed how to parse a CSV file.

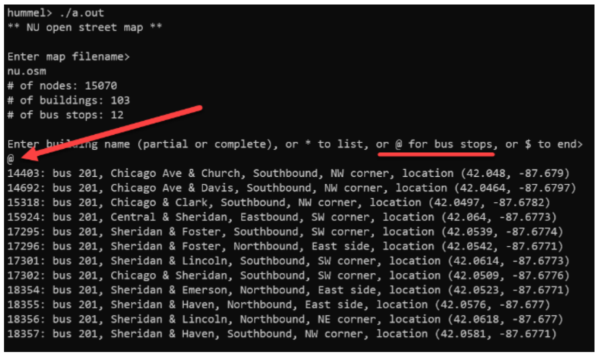
NOTE: these classes are a requirement of the assignment, along with the design of the BusStops constructor.

NOTE: the .h files should start with **`#pragma once`** to prevent “redefinition” errors when #included.

Step 04: printing bus stops

Recall that if the user enters the “*” for a building name, the program lists all the buildings. Extend the program so that if the user enters “@”, all the bus stops are listed in order by Stop ID.

Follow the design of adding **print( )** methods to the BusStop and BusStops classes. [This is a requirement of the assignment; no nested loops.]

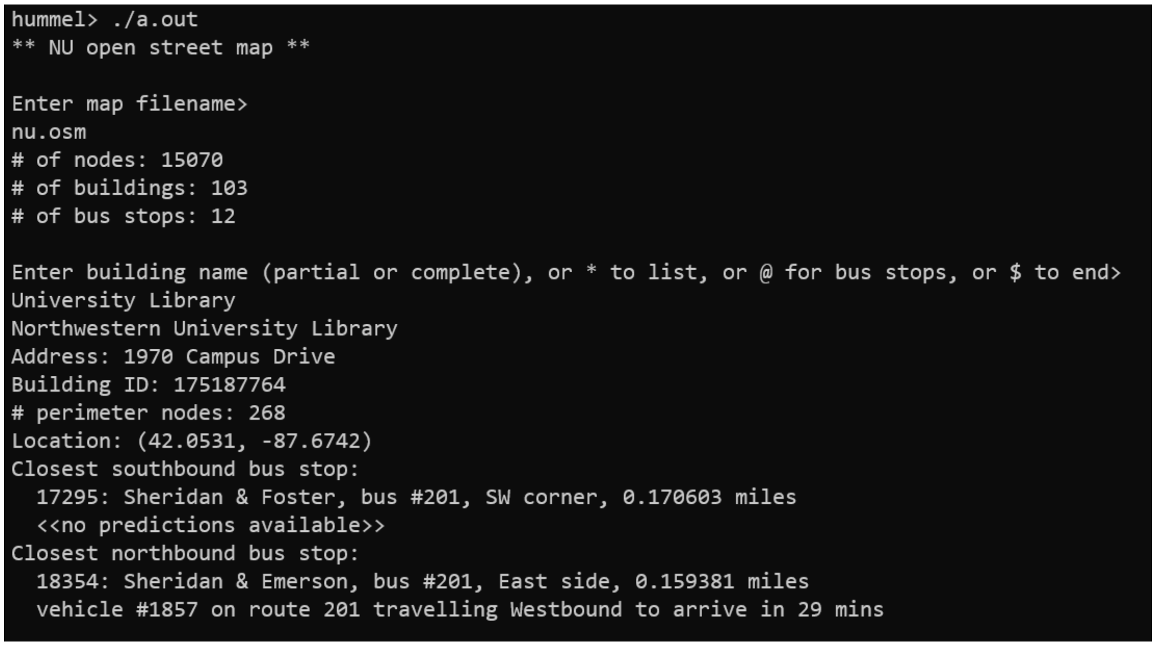


```
hummel> ./a.out
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of bus stops: 12

Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
@
14403: bus 201, Chicago Ave & Church, Southbound, NW corner, location (42.048, -87.679)
14692: bus 201, Chicago Ave & Davis, Southbound, NW corner, location (42.0464, -87.6797)
15318: bus 201, Chicago & Clark, Southbound, NW corner, location (42.0497, -87.6782)
15924: bus 201, Central & Sheridan, Eastbound, SW corner, location (42.064, -87.6773)
17295: bus 201, Sheridan & Foster, Southbound, SW corner, location (42.0539, -87.6774)
17296: bus 201, Sheridan & Foster, Northbound, East side, location (42.0542, -87.6771)
17301: bus 201, Sheridan & Lincoln, Southbound, SW corner, location (42.0614, -87.6773)
17302: bus 201, Chicago & Sheridan, Southbound, SW corner, location (42.0509, -87.6776)
18354: bus 201, Sheridan & Emerson, Northbound, East side, location (42.0523, -87.6771)
18355: bus 201, Sheridan & Haven, Northbound, East side, location (42.0576, -87.677)
18356: bus 201, Sheridan & Lincoln, Northbound, NE corner, location (42.0618, -87.677)
18357: bus 201, Sheridan & Haven, Southbound, NW corner, location (42.0581, -87.6771)
```

Step 05: closest bus stops

Now that you have access to the bus stop data, you can compute the closest bus stops to a given building. In particular, your task is to find the closest Southbound stop and closest Northbound stop. Match against the bus's direction of travel, which is one of four values: "Eastbound", "Northbound", "Southbound", or "Westbound". You may assume the input file "bus-stops.txt" contains at least one Southbound stop and one Northbound stop.

The file "dist.h" declares a function to find the distance (in miles) between two positions; #include this header file and use this function to help you. Note that the stops maybe different, e.g. from "University Library" the closest stops are "Sheridan & Foster" (Southbound) and "Sheridan & Emerson" (northbound):



```
hummel> ./a.out
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of bus stops: 12

Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
University Library
Northwestern University Library
Address: 1970 Campus Drive
Building ID: 175187764
# perimeter nodes: 268
Location: (42.0531, -87.6742)
Closest southbound bus stop:
  17295: Sheridan & Foster, bus #201, SW corner, 0.170603 miles
  <<no predictions available>>
Closest northbound bus stop:
  18354: Sheridan & Emerson, bus #201, East side, 0.159381 miles
  vehicle #1857 on route 201 travelling Westbound to arrive in 29 mins
```

There are various ways to solve this, one approach is to compute the distance to every stop, sort, and then select the closest one. You are free to solve as you see fit, with the following requirements:

1. *The code to compute the closest stops cannot be in main( ) nor in "main.cpp"*
2. *No explicit, nested loops*

Suggestion? Add a method to your BusStops class to return the closest stops to a given location.

Step 06: bus arrival predictions

As you probably know, the CTA has online bus and L trackers to help customers plan travel. We're going to use the freely-available CTA online bus tracker API to obtain bus predictions for the closest stops. Here's how to get started:

1. Follow the instructions here on "How to get an API key" for the online bus tracker API

2. Skim through the bus tracker API documentation here; in the table of contents, click on “Predictions” to read more about bus predictions. It’s okay if most of this doesn’t make sense, skim just to get a sense of what’s going on

The idea is that the CTA provides a server on the internet that answers web requests for data: you “call” the server using a carefully constructed URL, and the server will answer with the requested data (not a web page). Here’s an example from the documentation. Consider this URL:

http://ctabustracker.com/bustime/api/v2/getpredictions?key=89dj2he89d8j3j3ksjhdue93j&rt=20&stpid=456&format=json

Notice the underlined sections: these are the parts you supply to obtain the requested information: your API key, the bus route, the bus stop id, and the format of the data (use “json”). In response, the web server returns data in JSON format (*JavaScript Object Notation*) with 0 or more predictions for this bus route and stop:

```
{ "bustime-response":
  { "prd":
    [
      { "rid": "7028",
        "tripid": "57217175",
        "schdtm": "20130104 15:08",
        "tmstmp": "20130104 15:00",
        "typ": "A",
        "stpnm": "Madison & Jefferson",
        "stpid": "456",
        "vid": "2092",
        "dstp": "891",
        "rt": "20",
        "rtdir": "Westbound",
        "des": "Austin",
        "prdtm": "20130104 15:08",
        "tablockid": "87 -706",
        "tatripid": "1007569",
        "dly": false,
        "prdctdn": "8",
        "zone": ""
      }
    ]
  }
}
```

What you get back (as shown above) is a **map** { … } of (key, value) pairs. In this case, there’s only one key “bustime-response”, and the value is a nested map { “prd” : […] }. The “prd” standards for “predictions”, and the […] denote a list of maps. In the example above, there is only one map in the list:

```
    [
      { "rid": "7028",
        "tripid": "57217175",
        "schdtm": "20130104 15:08",
        "tmstmp": "20130104 15:00",
        "typ": "A",
        "stpnm": "Madison & Jefferson",
        "stpid": "456",
        "vid": "2092",
        "dstp": "891",
        "rt": "20",
        "rtdir": "Westbound",
        "des": "Austin",
        "prdtm": "20130104 15:08",
        "tablockid": "87 -706",
        "tatripid": "1007569",
        "dly": false,
        "prdctdn": "8",
        "zone": ""
      }
    ]
```

The important (key, value) pair we are looking for is “prdctdn”, this is the predicted time in minutes of when this bus --- denoted by “vid” for vehicle id --- is set to arrive at this stop. In the example above, the predicted time is 8 minutes. [*If you want to learn more about how this API works, take CS 310.*]

Here are the steps to make this happen:

1. Modify main( ) to initialize the CURL library we are using for internet communication:

   ```
   CURL* curl = curl_easy_init();
   if (curl == nullptr) {
     cout << "**ERROR:" << endl;
     cout << "**ERROR: unable to initialize curl library" << endl;
     cout << "**ERROR:" << endl;
     return 0;
   }
   ```

   You will also need to #include “curl_util.h”.

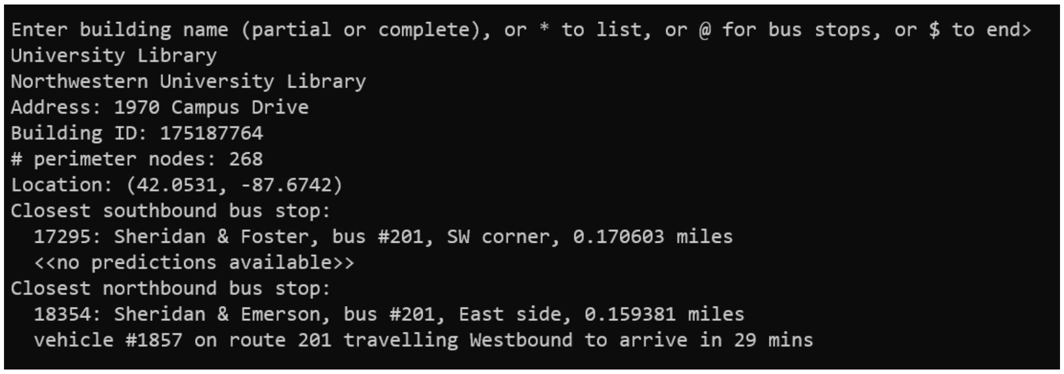
2. Your main( ) function should end by closing the CURL library, and then outputting “** Done **” as follows:

   ```
     //
     // done:
     //
     curl_easy_cleanup(curl);

     cout << endl;
     cout << "** Done **" << endl;

     return 0;

   }
   ```

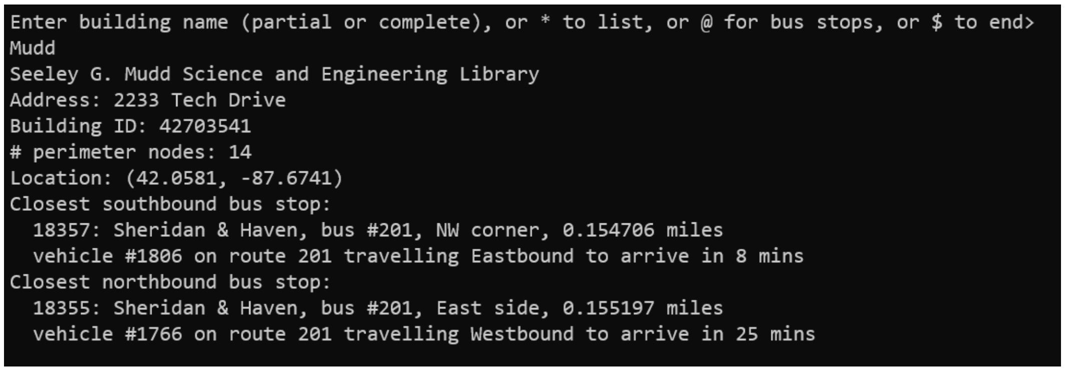
3. Find the closest bus stop (step 05)

4. Build string-based URL with API key, bus route number, and bus stop id. Use to_string( ) to convert integers to strings (route number and stop id).
5. Call the CTA online bus tracker API using the callWebServer( ) function declared in "curl_util.h".
6. If false is returned, output the error message "<<bus predictions unavailable, call failed>>" with an indentation of 2 spaces
7. If true is returned, extract the prediction(s) and output after the bus stop; each prediction should be indented 2 spaces. If no predictions are available (i.e. the JSON list is empty), output "<<no predictions available>>" with an indentation of 2 spaces.

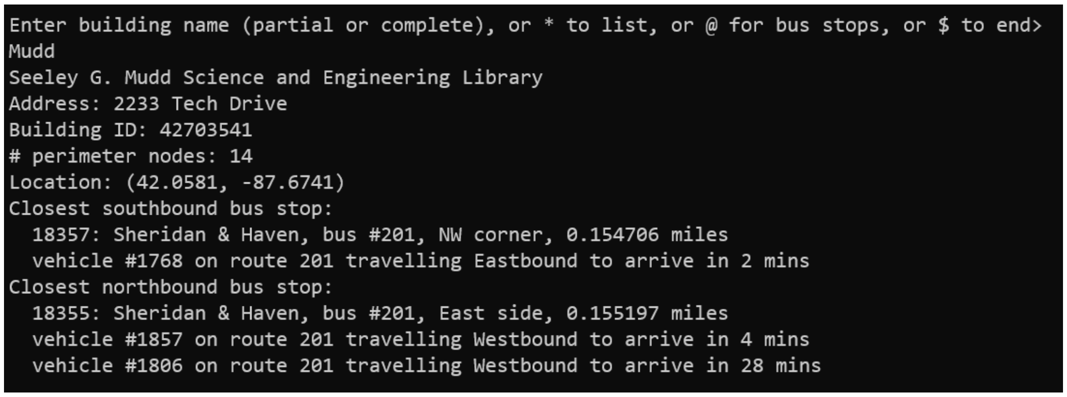
Here are examples of the various cases of no predictions, 1 prediction, and 2 predictions:



```
Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
University Library
Northwestern University Library
Address: 1970 Campus Drive
Building ID: 175187764
# perimeter nodes: 268
Location: (42.0531, -87.6742)
Closest southbound bus stop:
  17295: Sheridan & Foster, bus #201, SW corner, 0.170603 miles
  <<no predictions available>>
Closest northbound bus stop:
  18354: Sheridan & Emerson, bus #201, East side, 0.159381 miles
  vehicle #1857 on route 201 travelling Westbound to arrive in 29 mins
```



```
Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
Mudd
Seeley G. Mudd Science and Engineering Library
Address: 2233 Tech Drive
Building ID: 42703541
# perimeter nodes: 14
Location: (42.0581, -87.6741)
Closest southbound bus stop:
  18357: Sheridan & Haven, bus #201, NW corner, 0.154706 miles
  vehicle #1806 on route 201 travelling Eastbound to arrive in 8 mins
Closest northbound bus stop:
  18355: Sheridan & Haven, bus #201, East side, 0.155197 miles
  vehicle #1766 on route 201 travelling Westbound to arrive in 25 mins
```



```
Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
Mudd
Seeley G. Mudd Science and Engineering Library
Address: 2233 Tech Drive
Building ID: 42703541
# perimeter nodes: 14
Location: (42.0581, -87.6741)
Closest southbound bus stop:
  18357: Sheridan & Haven, bus #201, NW corner, 0.154706 miles
  vehicle #1768 on route 201 travelling Eastbound to arrive in 2 mins
Closest northbound bus stop:
  18355: Sheridan & Haven, bus #201, East side, 0.155197 miles
  vehicle #1857 on route 201 travelling Westbound to arrive in 4 mins
  vehicle #1806 on route 201 travelling Westbound to arrive in 28 mins
```

<< continued on next page >>

How to parse the JSON data returned by **callWebServer( )**? We are providing a JSON library in the header file "json.hpp". To use this library, the first step is to include the header file and import the namespace as follows:

```
#include "json.hpp"
using json = nlohmann::json;
```

After calling the web server, you parse the returned data and build a JSON object:

```
auto jsondata = json::parse(response);
```

There's one (key, value) pair under the key "bustime-response". Extract the value as follows:

```
auto bus_response = jsondata["bustime-response"];
```

The value **bus_response** is another map. The key is "prd", and the value is a list of 0 or more individual predictions: { "prd" : [{…}, {…}, …] }. At this point we want to loop through this list, which we can do using a "foreach" loop:

```
auto predictions = bus_response["prd"];

// for each prediction (a map) in the list:
for (auto& M : predictions) {

   cout << … << endl;
}
```

Read the loop as "foreach map M in the list of predictions". This implies that M is a map referencing one of the predictions. Recall a prediction looks like the following:

```
{ "rid": "7028",
  .
  .
  .
  "vid": "2092",
  "rt": "20",
  "rtdir": "Westbound",
  .
  .
  .
  "prdctdn": "8",
  "zone": ""
}
```

There are 4 values you need to extract and output to the console: "vid", "rt", "rtdir", and "prdctdn". For example, to output the vehicle id, you first extract the value from the JSON as a string, then convert to an integer for proper output:

```
cout << "  vehicle #" << stoi( M["vid"].get_ref<std::string&>() )
```

The above can all be nicely packaged into a function named *printBusPredictions( )*, which would naturally reside in the *BusStop* class. This is not a requirement, but makes good sense; the only requirements are this code cannot be a part of main( ), cannot appear in "main.cpp", and cannot contain nested loops.

What happens when you're running your program at night when no buses are operating? No predictions will be available, which is fine, but you'll have to run during the daytime to make sure your program is working. Alternatively, you can test your program with "mock" data; see Appendix A.

Step 07: exception handling

Dealing with real data, and calling other computer systems, is never perfect --- you have to expect errors to occur. In the case of the CTA bus predictions, you might get back an empty prediction "", or you might get "DUE" which means the bus is approaching the stop. How to handle things like this? In general, what's the best approach for handling "exceptional" situations in software?

Modern programming languages like C++, C#, Java and Python provide *exception handling* for dealing with errors / exceptional circumstances. Here in project 07, the place where you are going to encounter errors is the loop iterating through the predictions and outputting information:

```
// for each prediction in the list:
for (auto& M : predictions) {

    cout << … << endl;

}
```

If one of the keys is missing from the map M, or if the value is empty / invalid, then the output code inside the loop will fail --- triggering an often cryptic error message and the end of the program. To avoid this, what you want to do is "catch" any error that might occur and print an error message. This provides two advantages: (1) enables the output of a better error message, and (2) allows the program to keep running. First, #include the header file **<stdexcept>**. Second, add a try-catch statement inside the loop as follows:

```
for (auto& M : predictions) {

  try {
    cout << ... << endl;
  }
  catch (exception& e) {
    cout << "  error" << endl;
    cout << "  malformed CTA response, prediction unavailable"
         << " (error: " << e.what() << ")" << endl;
  }

}
```

Exception handling is a hallmark of well-written software.

Grading and Electronic Submission

You will submit all your .cpp and .h files to Gradescope for evaluation. To submit from the EECS computers, run the following command (you must run this command from inside your **project07/release** directory):

```
/home/cs211/w2024/tools/project07  submit  *.cpp  *.h  *.hpp
```

Gradescope submissions will open 2-3 days before the due date; you'll have 4 submissions per 24-hr period.

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your "Submission History". This must be done before the due date.

The autograding portion of the project will be worth 80/100 points. The remaining 20 points will be determined manually by confirming the following requirements, reviewing your approach, and commenting. However, you may lose correctness points if you have a feature correctly implemented, but manual review reveals that the proper steps were not taken (for example, not providing the required classes, or not adhering to the rule of "no explicit, nested loops"):

1. *Implementing required building location (step 02)*
2. *Implementing required BusStop and BusStops classes (step 03)*
3. *Requirements for closest bus stops (step 05)*
4. *Requirements for bus predictions (step 06)*
5. *Exception handling (step 07)*
6. *Header comments at top of each .h and .c file, and above each function*
7. *Meaningful comments describing blocks of code (e.g. loops)*
8. *Valgrind errors / memory leaks? In this project you can ignore any memory leaks, the CURL library we are using appears to leak memory, at least the way we are using it. You can safely ignore all memory leaks reported by valgrind.*

Omission of item 1, or 2, or 3, or 4 will result in an overall score of 0 for the assignment; you must meet the requirements implied by items 1 – 4 above.

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found here. In summary, here are NU's eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*

8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU's academic integrity website. With regards to CS 211, unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk and learn from the course staff, peers, and others, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

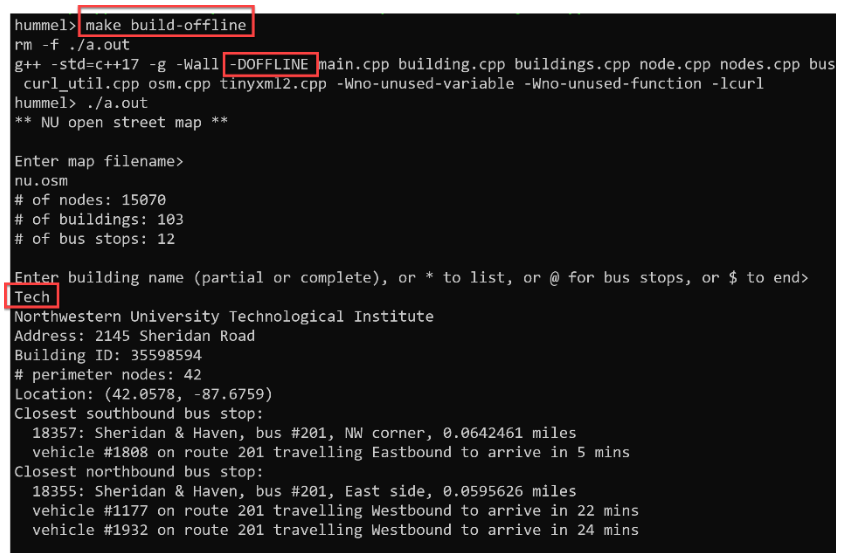
Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.

Appendix: working offline ("mocking")

An interesting problem arises when building software that interacts with online servers like the CTA online bus tracker API: what happens when the online system is not working? Or what happens at night when the buses are not running? In general, the problem that arises is how to test our software when the online system is not available.

The solution is to work with "fake" data when an online system is not available. In software engineering terms, this is called "mocking" --- creating a simulated object / data that acts in place of a real object / live data. In the context of this project, we "mock" the online data inside the ***callWebServer( )*** function defined in "curl_util.cpp". Go ahead and open this file in your editor… You'll see this structure to the function:

```
bool callWebServer(CURL* curl, string url, string& response)
{

#ifdef OFFLINE
  //
  // We are offline:
  //
  .
  .
  .
#else
  //
  // We are online:
  //
  .
  .
  .
#endif

}
```



```
hummel> make build-offline
rm -f ./a.out
g++ -std=c++17 -g -Wall -DOFFLINE main.cpp building.cpp buildings.cpp node.cpp nodes.cpp bus
 curl_util.cpp osm.cpp tinyxml2.cpp -Wno-unused-variable -Wno-unused-function -lcurl
hummel> ./a.out
** NU open street map **

Enter map filename>
nu.osm
# of nodes: 15070
# of buildings: 103
# of bus stops: 12

Enter building name (partial or complete), or * to list, or @ for bus stops, or $ to end>
Tech
Northwestern University Technological Institute
Address: 2145 Sheridan Road
Building ID: 35598594
# perimeter nodes: 42
Location: (42.0578, -87.6759)
Closest southbound bus stop:
  18357: Sheridan & Haven, bus #201, NW corner, 0.0642461 miles
  vehicle #1808 on route 201 travelling Eastbound to arrive in 5 mins
Closest northbound bus stop:
  18355: Sheridan & Haven, bus #201, East side, 0.0595626 miles
  vehicle #1177 on route 201 travelling Westbound to arrive in 22 mins
  vehicle #1932 on route 201 travelling Westbound to arrive in 24 mins
```

When offline, you'll see that the function reads data from text files named "cta-response….cta"; these files contain data saved from earlier, online sessions. The use of **#ifdef** is common in C/C++ as a way to compile your program for different scenarios. In this case we have two versions of the function: one for *offline* behavior, and one for *online* behavior. You then compile the program differently based on which version you want: compile with the option -DOFFLINE when you want offline behavior, and compile without this option when you want online behavior (this becomes the default). Based on the provided makefile, use "make build" to compile for online behavior, and use "make build-offline" when you want to compile for offline behavior. Then run "./a.out" as normal. The screenshot above is an example of what you should see when you run OFFLINE and search for the Tech building.

If you want to experiment with this approach, open a terminal connection to an EECS computer and copy the updated files (note that this will change your "makefile", so if you have edited that file make a copy first):

1. Open a terminal window in VS Code. Or, if you are working on a Mac or Linux, open a terminal; if you are working on Windows, you can open Git Bash.
2. Login to moore: **`ssh moore`** or **`ssh YOUR_NETID@moore.wot.eecs.northwestern.edu`**
3. Move ("change") into this release dir **`cd project07/release`**
4. Copy the updated files: **`cp /home/cs211/w2024/project07/mock/* .`**
5. List the contents of the directory **`ls`**

Once you have copied the files, build your program as usual with "**make build**". When you want to work in OFFLINE mode, use "**make build-offline**". Finally, if you want to create your own offline data files, build your program using "**make build-online-save**" and run --- your online data will be saved to text files for future, offline processing.