Extra Credit Project: Covid-19 data analysis program

Complete By: Sunday, March 29th @ 11:59pm

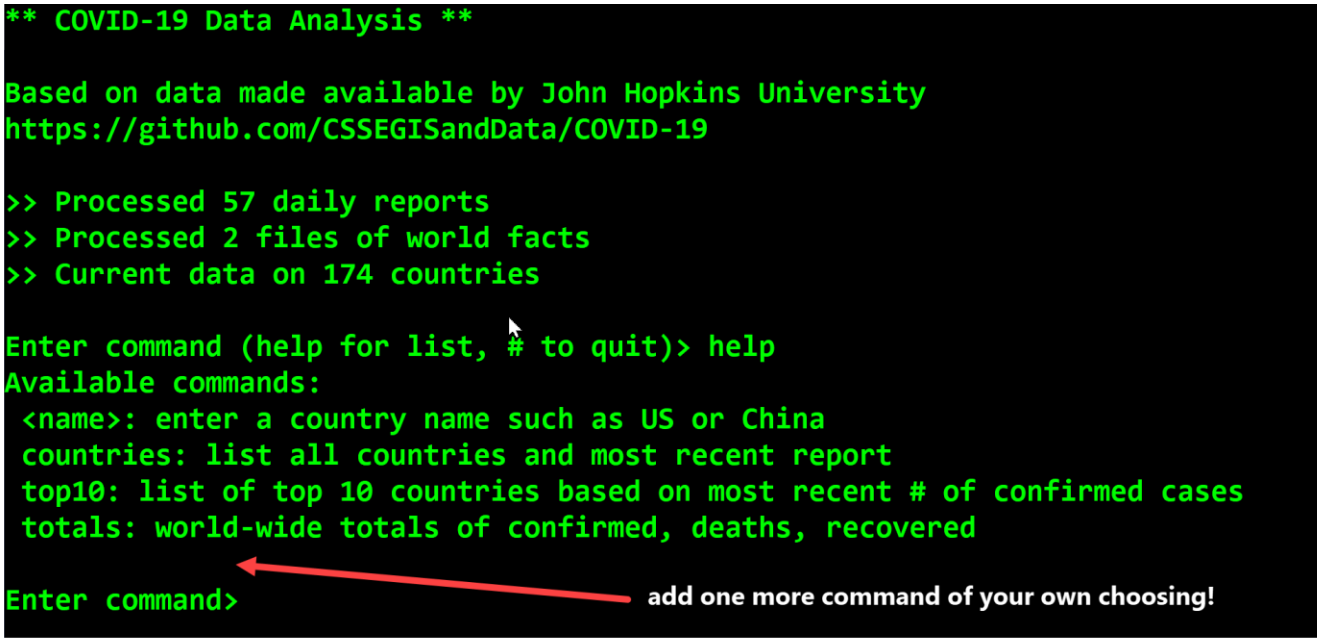
Assignment: C++ program to analyze Covid-19 daily reports

Policy: Individual work only, late work is *not* accepted

Submission: "main.cpp" file + additional files via Gradescope

Assignment

The assignment is to use a C++ **map** data structure (and other data structures if needed) to input and analyze daily reports surrounding the Covid-19 virus. Since this is an extra credit project, it's an all-or-nothing proposition: you must score 100 via the autograder, you must meet all requirements, no partial credit, etc. This is not meant to be a hard project, in fact it's a good project that puts into practice all the things we've been working on: data structures, functional design, encapsulation, file input. But we don't have a lot of resources to devote to grading, so be sure to follow the project and autograding requirements. Here's the main screen for the program:



```
** COVID-19 Data Analysis **

Based on data made available by John Hopkins University
https://github.com/CSSEGISandData/COVID-19

>> Processed 57 daily reports
>> Processed 2 files of world facts
>> Current data on 174 countries

Enter command (help for list, # to quit)> help
Available commands:
 <name>: enter a country name such as US or China
 countries: list all countries and most recent report
 top10: list of top 10 countries based on most recent # of confirmed cases
 totals: world-wide totals of confirmed, deaths, recovered

Enter command>
```

In short, the program supports 6 commands:

1. **help**
2. Lookup and display of data by country **<name>**

3. List all **countries** in alphabetical order with data from most recent daily report
4. List the **top10** countries based on # of confirmed cases
5. List the world-wide **totals** (confirmed, deaths, recovered)
6. A command of your own choice

We would encourage folks to post what they are doing for #6 on Piazza, to generate more ideas. Lots and lots of possibilities.

Programming Environment

You are free to program on whatever platform you want, using whatever C++ compiler / programming environment you want. We are providing a Codio project **cs251-extra-credit—covid-19** containing the input files and a starter "main.cpp" file; these are also provided on the course dropbox. Note that the provided "main.cpp" requires C++17; see page 6 of this document for more info on enabling C++17.

Input Files and Processing

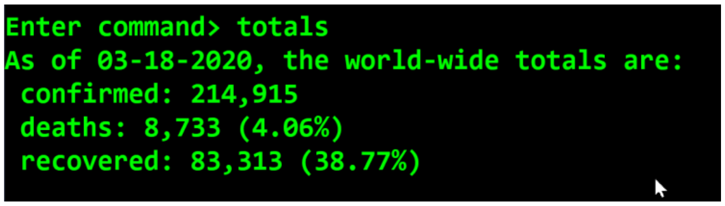
The input files are all .csv files (comma-separated files). If you double-click they will open in Excel, which is not that helpful. Better to open in a text editor so you can see what the file actually looks like. There are two sets of files. The first set are daily reports put together by John Hopkins university. Keep in mind this is real data, and real data is often messy. Example: the names of countries are constantly changing, such as "Mainland China" in the early reports is now simply referred as "China". Here's an example of the first daily report file named "01-22-2020.csv":

```
Province/State,Country/Region,Last Update,Confirmed,Deaths,Recovered
Anhui,Mainland China,1/22/2020 17:00,1,,
Beijing,Mainland China,1/22/2020 17:00,14,,
Chongqing,Mainland China,1/22/2020 17:00,6,,
.
.
.
,Japan,1/22/2020 17:00,2,,
,Thailand,1/22/2020 17:00,2,,
,South Korea,1/22/2020 17:00,1,,
```

The first row is a header row; input that line and discard it. The remaining rows are the data rows, and the total # of data rows is unpredictable. Notice that some rows do not have a province/state, so the first column may be empty. And right now most of the last 2 columns are empty --- there are no deaths or recovered to report. For missing confirmed, deaths, or recovered, use 0. The date of the report can be obtained from the filename, or from the last update column; I prefer the filename because it has the leading 0 (better been ordering by date).

For this daily report, you need to accumulate the data by country --- i.e. ignore the province/state, and focus on the country. For each country, sum the confirmed cases, the deaths, and the recovered cases. Store all 3 sums in a map, using the country name as the key for fast lookup and data accumulation. You can use 3 different maps, but I would recommend a single map where the country is the key, and the value is a struct that contains the country name and the 3 sums. Example: on “01-22-2020”, China had a total of 547 confirmed cases, 17 deaths, and 28 recovered cases. On “03-18-2020”, China had a total of 81,102 confirmed cases, 3,241 deaths, and 69,755 recovered cases.

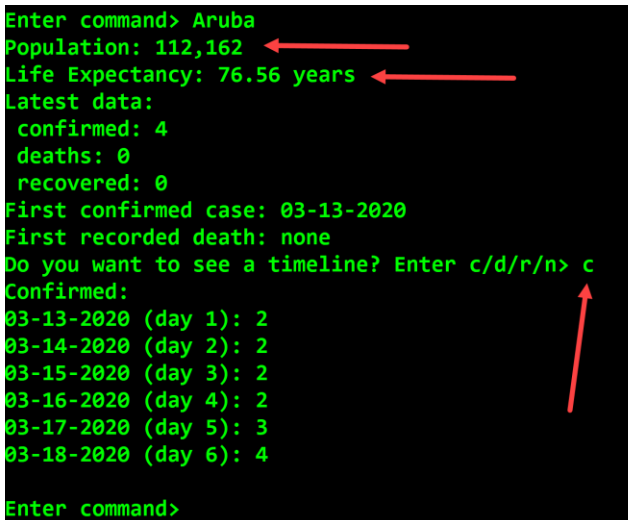
There are currently 57 daily report files, which are cleaned up and posted on the course drop box under “extra-credit-files”; the raw data can be downloaded from https://github.com/CSSEGISandData/COVID-19 (using this tool if need be: https://desktop.github.com/). Each daily report includes the previous, i.e. the report on “01-23-2020” represents the new total # of confirmed, deaths, and recovered cases. And then “01-24-2020” includes that one, and so on for each daily report. Thus, the last report on “03-18-2020” contains the world-wide totals for confirmed cases, deaths, and recovered cases. This is what your “totals” command should output, the totals from this last report:



```
Enter command> totals
As of 03-18-2020, the world-wide totals are:
 confirmed: 214,915
 deaths: 8,733 (4.06%)
 recovered: 83,313 (38.77%)
```

The implication is that your program needs to store these 3 sums --- confirmed, deaths, and recovered --- for each country, and for each of the 57 dates. And you need to do this for the general case, i.e. assume N daily reports where N could be in the thousands. For testing, we’ll release each new daily report as it becomes available, so you can test your program to make sure it’s not tied to the value of 57. [Suggestion: use a single map where the key is the country name, and the value is not just a name and 3 sums, but now it’s a name and 1 or more additional data structures to store all the sums for that country. A map of maps? A map of vectors? Since we are accumulating data by country and also date, you want to support fast lookup. So the most natural data structure to use in these cases is… This decision will impact the entire program, both in terms of programming, and efficiency.]

To drive this point home a bit more, we have 2 additional “world facts” files to input and merge with your country data: populations and life expectancies: “populations.csv” and “life_expectancies.csv”, respectively. We’ll let you open and explore these files, but they can also be found on the course dropbox. These files should be input and stored by country along with your other data. This information is output when the user enters a country name, such as **Aruba**:



```
Enter command> Aruba
Population: 112,162
Life Expectancy: 76.56 years
Latest data:
 confirmed: 4
 deaths: 0
 recovered: 0
First confirmed case: 03-13-2020
First recorded death: none
Do you want to see a timeline? Enter c/d/r/n> c
Confirmed:
03-13-2020 (day 1): 2
03-14-2020 (day 2): 2
03-15-2020 (day 3): 2
03-16-2020 (day 4): 2
03-17-2020 (day 5): 3
03-18-2020 (day 6): 4

Enter command>
```

Note that there may be countries in the daily reports that do not appear in the world facts files, e.g. "Reunion" or "Cruise Ship". Likewise, there are countries in the world facts files that do not exist in the daily reports, e.g. "European Union". This is typical with real data. If you lookup a world fact for country X and that value cannot be found, use 0.

How to work with .csv files? Do not download and install outside libraries, since those will not be available when you submit to gradescope. It's not too hard to do everything you need from within C++. First, open and input from a .csv file like any other text file: use an **ifstream** object, and check to make sure the file was successfully opened (output an error msg and skip the file if it cannot be opened). Use the **getline()** function to read line-by-line, skipping the first line (header row). Here's the loop, assuming **infile** is your file variable:

```
string line;

getline(infile, line);  // skip first line (header row):

while (getline(infile, line))
{
  .
  .
  .
}
```

Now, how to process each line? This is the messy part --- all the code that follows goes *inside* the while loop. First, we have to deal with the fact that some of the file contain extra commas, in particular to deal with US cities and dates (e.g. "Chicago, IL"). So some of the lines start with ", and if so, we have to change that from say "City, State" to just City State:

```
if (line[0] == '"')  // => province is "city, state"
{
  //
  // we want to delete the " on either end of the value, and
  // delete the ',' embedded within => will get city state:
  //
  line.erase(0, 1);      // delete the leading "
  size_t pos = line.find(',');  // find embedded ','
  line.erase(pos, 1);   // delete ','
  pos = line.find('"'); // find closing "
  line.erase(pos, 1);   // delete closing "
}
```

Okay, now the line has the correct number of commas, and we can start to parse the data into its individual parts. We use the C++ **stringstream** class to help:

```
stringstream s(line);
string province, country, last_update;
string confirmed, deaths, recovered;

getline(s, province, ',');
getline(s, country, ',');
getline(s, last_update, ',');
getline(s, confirmed, ',');
```

```
    getline(s, deaths, ',');
    getline(s, recovered, ',');

    if (confirmed == "")
      confirmed = "0";
    if (deaths == "")
      deaths = "0";
    if (recovered == "")
      recovered = "0";

    if (country == "Mainland China")  // map to name in the earlier reports:
      country = "China";
```

At this point, you have the values you need to store / accumulate in the **map**. Again, use the country as the key so you can merge this data efficiently with any existing data. If you forget how to use a **map**, there is a section later in this document to summarize the map API. Use a similar approach to input the data from the world facts files.

WRITE A SINGLE FUNCTON to process a daily report. And then call this function N times, once for each report. If you write N separate functions to input the reports, you will receive a 0 on the project (and you may even fail the class :-). Seriously, you are better than that.

To help you input the daily reports, we are going to use a class that was made available in C++17: **filesystem**. Here's a function that given a folder / directory name, returns the names of all the files in that folder / directory:

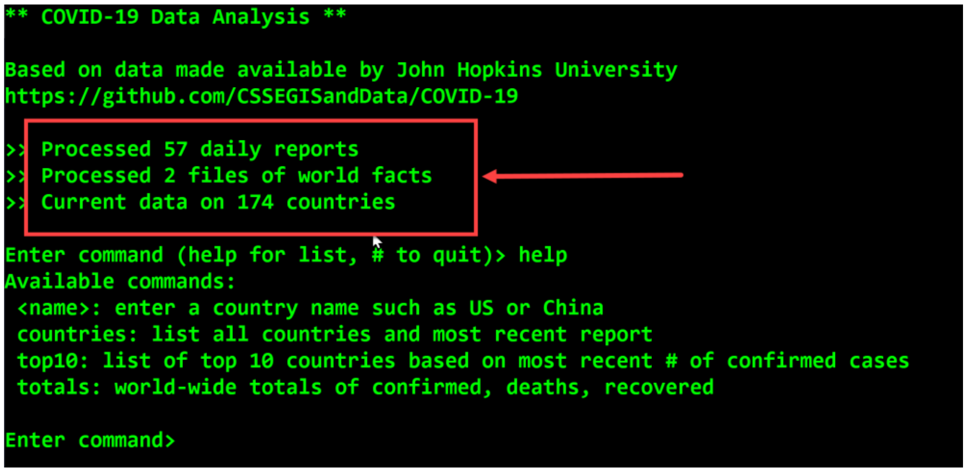
```
  //
  // getFilesWithinFolder
  //
  // Given the path to a folder, e.g. "./daily_reports/", returns
  // a vector containing the full pathnames of all regular files
  // in this folder.  If the folder is empty, the vector is empty.
  //
  vector<string> getFilesWithinFolder(string folderPath)
  {
    vector<string> files;

    for (const auto& entry : fs::directory_iterator(folderPath)) {
      if (entry.is_regular_file()) {
        // this gives you just the filename:
        // files.push_back(entry.path().filename().string());

        // this gives you the full path + filename:
        files.push_back(entry.path().string());
      }
    }

    // let's make sure the files are in alphabetical order, so we process in order by date:
    std::sort(files.begin(), files.end());

    return files;
  }
```

In your main() program, you'll call this function as follows:

```
vector<string> files = getFilesWithinFolder("./daily_reports/");

for (string filename : files)
{
  readOneDailyReport(filename, data);
}
```

where **data** is a map for accumulating the daily reports. If you are working outside Codio, you'll need to adjust your compiler settings to specify C++17. Using g++, you would use the compiler option -std=c++17. In Visual Studio, you must change a project property: Project menu, project properties (very bottom of menu?), expand C/C++, Language, change "C++ Language Standard" to "ISO C++ 17".

The course dropbox will contain a starter "main.cpp" that contains the necessary #include files, this filesystem-based function, and the beginning of main(). Once you have the input files processed, note that the program should output the following stats:



```
** COVID-19 Data Analysis **

Based on data made available by John Hopkins University
https://github.com/CSSEGISandData/COVID-19

>> Processed 57 daily reports
>> Processed 2 files of world facts
>> Current data on 174 countries

Enter command (help for list, # to quit)> help
Available commands:
 <name>: enter a country name such as US or China
 countries: list all countries and most recent report
 top10: list of top 10 countries based on most recent # of confirmed cases
 totals: world-wide totals of confirmed, deaths, recovered

Enter command>
```

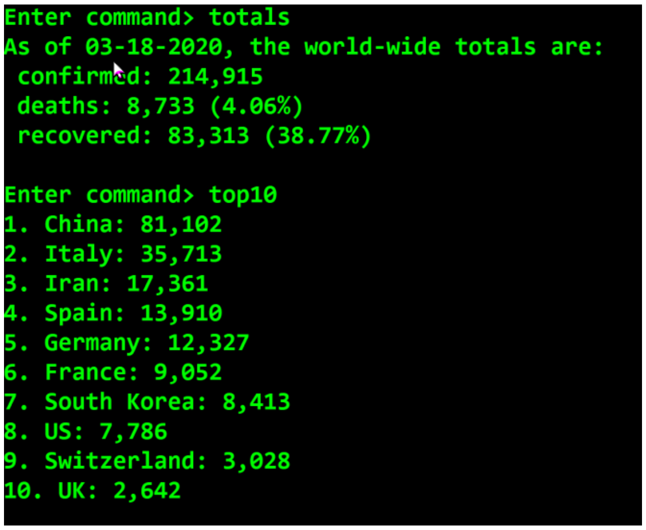
Program functionality

The program supports 6 commands, the last of which is a command of your choice. Since a country's name may consist of multiple words (e.g. South Korea), you should input commands from the user using **getline**(cin, command):

1. **help**
2. Lookup and display of data by country **<name>**
3. List all **countries** in alphabetical order with data from most recent daily report
4. List the **top10** countries based on # of confirmed cases
5. List the world-wide **totals** (confirmed, deaths, recovered)
6. A command of your own choice

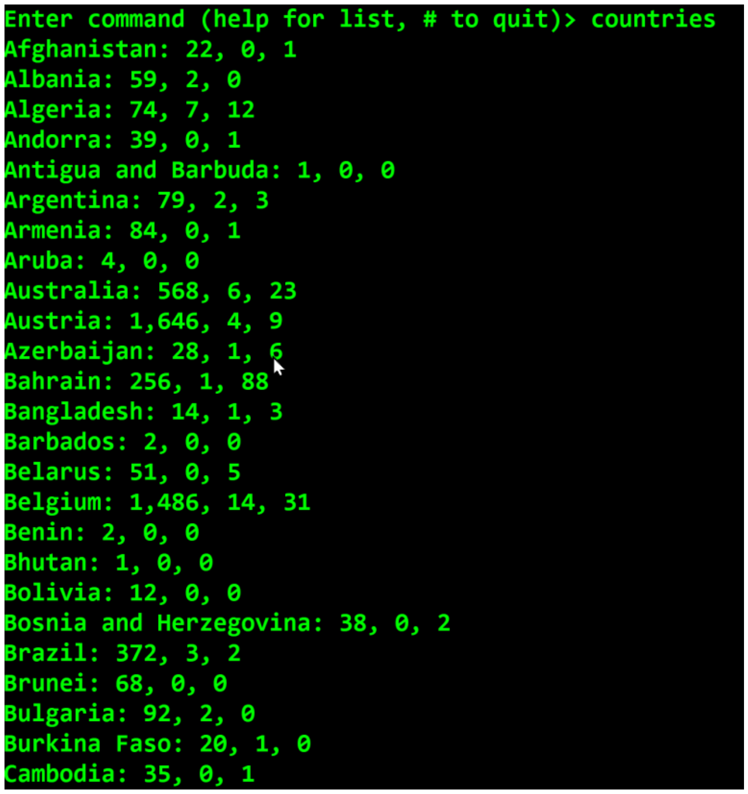
Please don't ask us what you should do for #6, it's up to you --- just do something non-trivial. We encourage you to post your ideas on Piazza to help generate additional ideas.

As of “03-18-2020”, here are the outputs for the **totals** and **top10** commands. Feel free to use the built-in sort algorithm if you want:

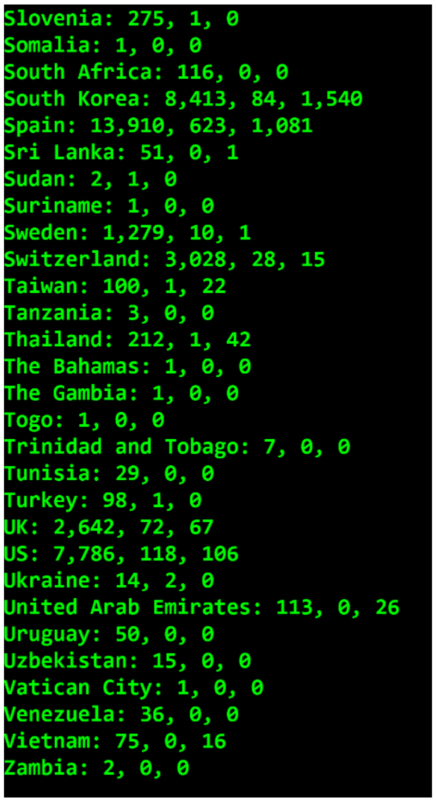


```
Enter command> totals
As of 03-18-2020, the world-wide totals are:
 confirmed: 214,915
 deaths: 8,733 (4.06%)
 recovered: 83,313 (38.77%)

Enter command> top10
1. China: 81,102
2. Italy: 35,713
3. Iran: 17,361
4. Spain: 13,910
5. Germany: 12,327
6. France: 9,052
7. South Korea: 8,413
8. US: 7,786
9. Switzerland: 3,028
10. UK: 2,642
```

As of “03-18-2020”, for the **countries** command there should be 174 countries output in alphabetical order. Here are the first and last 20+ countries:



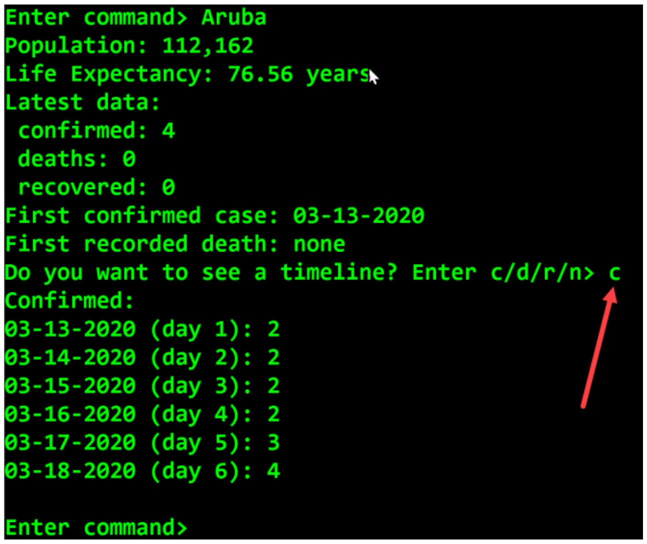
```
Enter command (help for list, # to quit)> countries
Afghanistan: 22, 0, 1
Albania: 59, 2, 0
Algeria: 74, 7, 12
Andorra: 39, 0, 1
Antigua and Barbuda: 1, 0, 0
Argentina: 79, 2, 3
Armenia: 84, 0, 1
Aruba: 4, 0, 0
Australia: 568, 6, 23
Austria: 1,646, 4, 9
Azerbaijan: 28, 1, 6
Bahrain: 256, 1, 88
Bangladesh: 14, 1, 3
Barbados: 2, 0, 0
Belarus: 51, 0, 5
Belgium: 1,486, 14, 31
Benin: 2, 0, 0
Bhutan: 1, 0, 0
Bolivia: 12, 0, 0
Bosnia and Herzegovina: 38, 0, 2
Brazil: 372, 3, 2
Brunei: 68, 0, 0
Bulgaria: 92, 2, 0
Burkina Faso: 20, 1, 0
Cambodia: 35, 0, 1
```



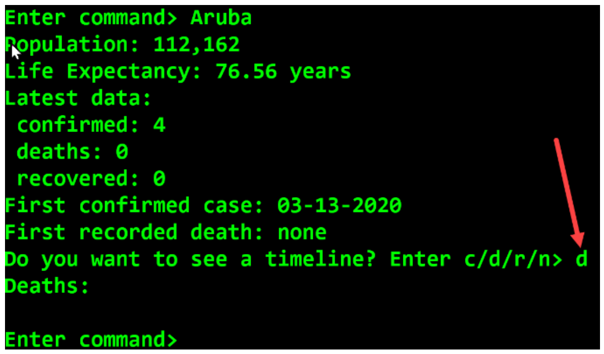
```
Slovenia: 275, 1, 0
Somalia: 1, 0, 0
South Africa: 116, 0, 0
South Korea: 8,413, 84, 1,540
Spain: 13,910, 623, 1,081
Sri Lanka: 51, 0, 1
Sudan: 2, 1, 0
Suriname: 1, 0, 0
Sweden: 1,279, 10, 1
Switzerland: 3,028, 28, 15
Taiwan: 100, 1, 22
Tanzania: 3, 0, 0
Thailand: 212, 1, 42
The Bahamas: 1, 0, 0
The Gambia: 1, 0, 0
Togo: 1, 0, 0
Trinidad and Tobago: 7, 0, 0
Tunisia: 29, 0, 0
Turkey: 98, 1, 0
UK: 2,642, 72, 67
US: 7,786, 118, 106
Ukraine: 14, 2, 0
United Arab Emirates: 113, 0, 26
Uruguay: 50, 0, 0
Uzbekistan: 15, 0, 0
Vatican City: 1, 0, 0
Venezuela: 36, 0, 0
Vietnam: 75, 0, 16
Zambia: 2, 0, 0
```

<< continued on next page >>

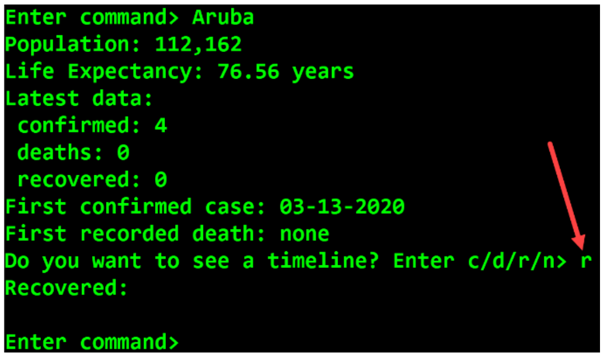
Finally, here are a few examples of individual countries. If the user enters a country that does not exist, output “country or command not found…”. Here’s some data on **Aruba**. The latest data comes from the most recent daily report file. The first confirmed case, and death, is obtained by searching the accumulated data for the first non-zero values (or “none”). Finally, there are 3 types of timelines: confirmed, deaths, and recovered. The user can also enter “n” for none. The timeline outputs the accumulated data for that country:



```
Enter command> Aruba
Population: 112,162
Life Expectancy: 76.56 years
Latest data:
 confirmed: 4
 deaths: 0
 recovered: 0
First confirmed case: 03-13-2020
First recorded death: none
Do you want to see a timeline? Enter c/d/r/n> c
Confirmed:
03-13-2020 (day 1): 2
03-14-2020 (day 2): 2
03-15-2020 (day 3): 2
03-16-2020 (day 4): 2
03-17-2020 (day 5): 3
03-18-2020 (day 6): 4

Enter command>
```

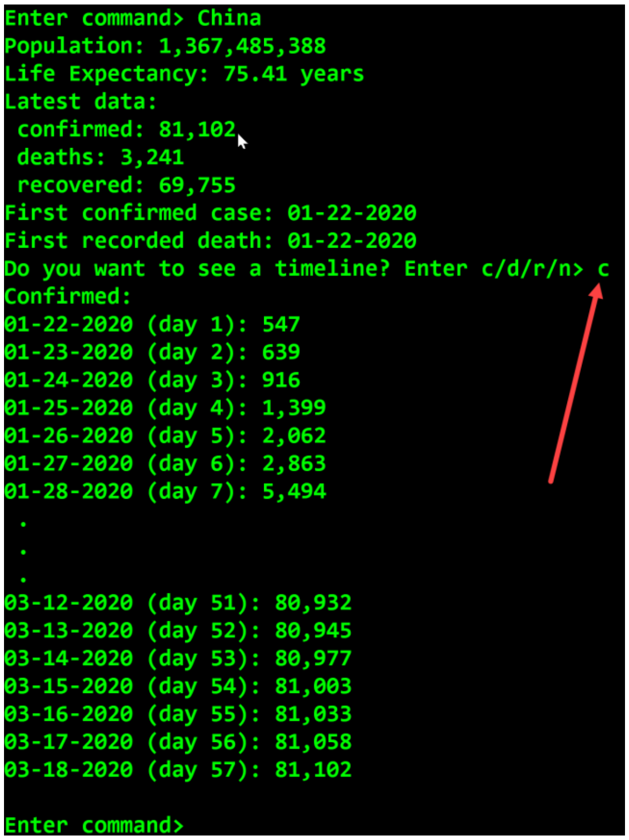


```
Enter command> Aruba
Population: 112,162
Life Expectancy: 76.56 years
Latest data:
 confirmed: 4
 deaths: 0
 recovered: 0
First confirmed case: 03-13-2020
First recorded death: none
Do you want to see a timeline? Enter c/d/r/n> d
Deaths:

Enter command>
```

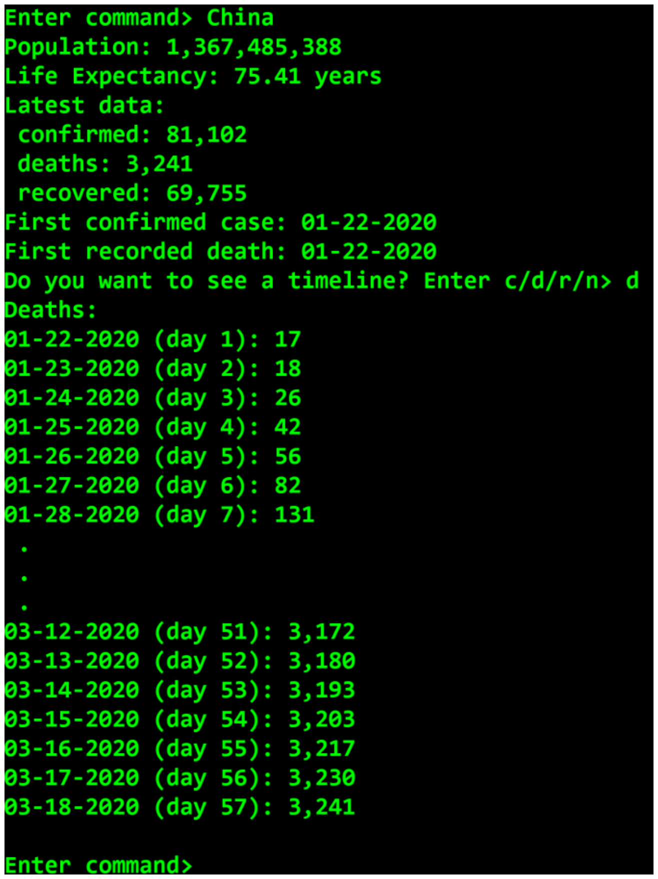


```
Enter command> Aruba
Population: 112,162
Life Expectancy: 76.56 years
Latest data:
 confirmed: 4
 deaths: 0
 recovered: 0
First confirmed case: 03-13-2020
First recorded death: none
Do you want to see a timeline? Enter c/d/r/n> r
Recovered:

Enter command>
```

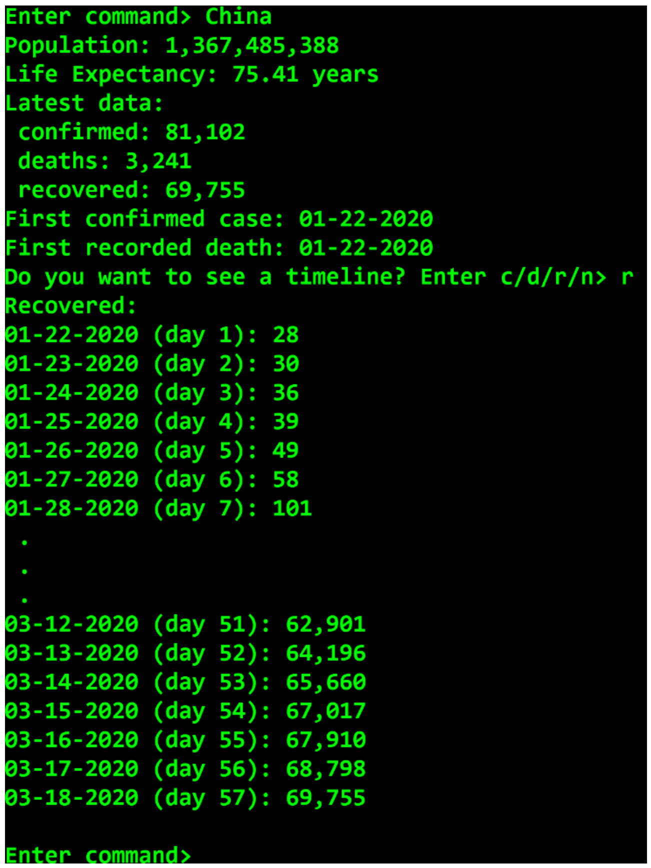
Here’s some data for **China**. In this case, the timelines show the first and last 7 reports:



```
Enter command> China
Population: 1,367,485,388
Life Expectancy: 75.41 years
Latest data:
 confirmed: 81,102
 deaths: 3,241
 recovered: 69,755
First confirmed case: 01-22-2020
First recorded death: 01-22-2020
Do you want to see a timeline? Enter c/d/r/n> c
Confirmed:
01-22-2020 (day 1): 547
01-23-2020 (day 2): 639
01-24-2020 (day 3): 916
01-25-2020 (day 4): 1,399
01-26-2020 (day 5): 2,062
01-27-2020 (day 6): 2,863
01-28-2020 (day 7): 5,494
 .
 .
 .
03-12-2020 (day 51): 80,932
03-13-2020 (day 52): 80,945
03-14-2020 (day 53): 80,977
03-15-2020 (day 54): 81,003
03-16-2020 (day 55): 81,033
03-17-2020 (day 56): 81,058
03-18-2020 (day 57): 81,102

Enter command>
```

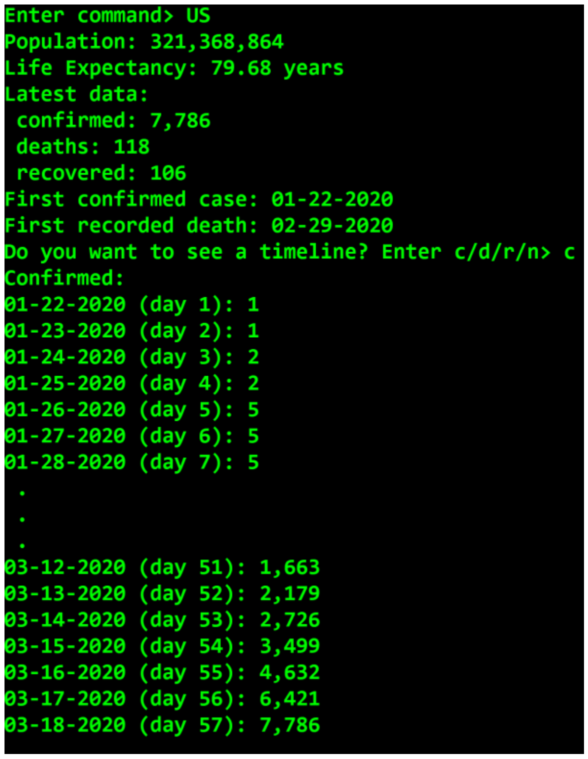


```
Enter command> China
Population: 1,367,485,388
Life Expectancy: 75.41 years
Latest data:
 confirmed: 81,102
 deaths: 3,241
 recovered: 69,755
First confirmed case: 01-22-2020
First recorded death: 01-22-2020
Do you want to see a timeline? Enter c/d/r/n> d
Deaths:
01-22-2020 (day 1): 17
01-23-2020 (day 2): 18
01-24-2020 (day 3): 26
01-25-2020 (day 4): 42
01-26-2020 (day 5): 56
01-27-2020 (day 6): 82
01-28-2020 (day 7): 131
 .
 .
 .
03-12-2020 (day 51): 3,172
03-13-2020 (day 52): 3,180
03-14-2020 (day 53): 3,193
03-15-2020 (day 54): 3,203
03-16-2020 (day 55): 3,217
03-17-2020 (day 56): 3,230
03-18-2020 (day 57): 3,241

Enter command>
```

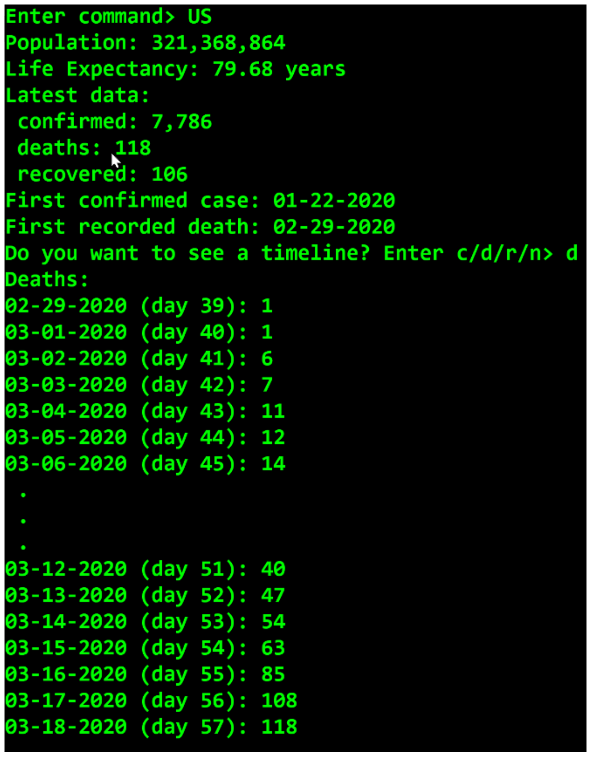


```
Enter command> China
Population: 1,367,485,388
Life Expectancy: 75.41 years
Latest data:
 confirmed: 81,102
 deaths: 3,241
 recovered: 69,755
First confirmed case: 01-22-2020
First recorded death: 01-22-2020
Do you want to see a timeline? Enter c/d/r/n> r
Recovered:
01-22-2020 (day 1): 28
01-23-2020 (day 2): 30
01-24-2020 (day 3): 36
01-25-2020 (day 4): 39
01-26-2020 (day 5): 49
01-27-2020 (day 6): 58
01-28-2020 (day 7): 101
 .
 .
 .
03-12-2020 (day 51): 62,901
03-13-2020 (day 52): 64,196
03-14-2020 (day 53): 65,660
03-15-2020 (day 54): 67,017
03-16-2020 (day 55): 67,910
03-17-2020 (day 56): 68,798
03-18-2020 (day 57): 69,755

Enter command>
```

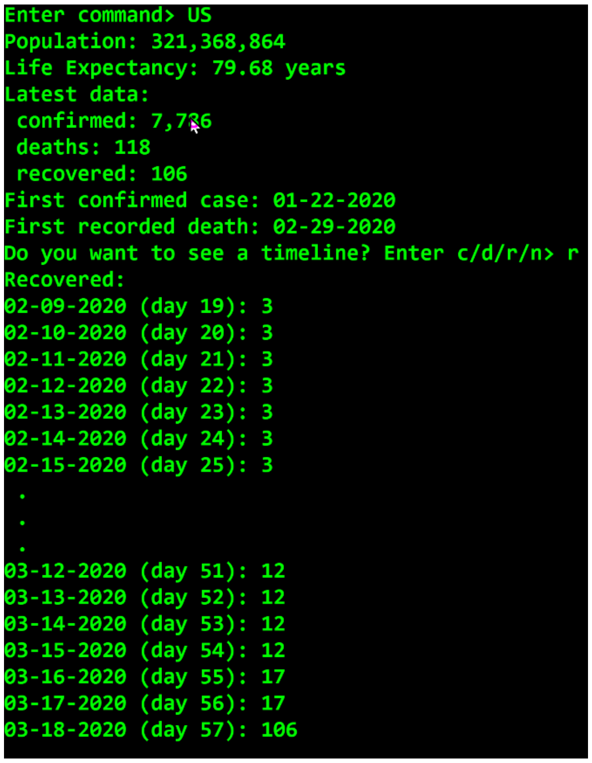
If a timeline consists of > 14 entries, then show the first and last 7, otherwise show the complete timeline. Finally, here's the data for the US as of "03-18-2020":



```
Enter command> US
Population: 321,368,864
Life Expectancy: 79.68 years
Latest data:
 confirmed: 7,786
 deaths: 118
 recovered: 106
First confirmed case: 01-22-2020
First recorded death: 02-29-2020
Do you want to see a timeline? Enter c/d/r/n> c
Confirmed:
01-22-2020 (day 1): 1
01-23-2020 (day 2): 1
01-24-2020 (day 3): 2
01-25-2020 (day 4): 2
01-26-2020 (day 5): 5
01-27-2020 (day 6): 5
01-28-2020 (day 7): 5
 .
 .
 .
03-12-2020 (day 51): 1,663
03-13-2020 (day 52): 2,179
03-14-2020 (day 53): 2,726
03-15-2020 (day 54): 3,499
03-16-2020 (day 55): 4,632
03-17-2020 (day 56): 6,421
03-18-2020 (day 57): 7,786
```



```
Enter command> US
Population: 321,368,864
Life Expectancy: 79.68 years
Latest data:
 confirmed: 7,786
 deaths: 118
 recovered: 106
First confirmed case: 01-22-2020
First recorded death: 02-29-2020
Do you want to see a timeline? Enter c/d/r/n> d
Deaths:
02-29-2020 (day 39): 1
03-01-2020 (day 40): 1
03-02-2020 (day 41): 6
03-03-2020 (day 42): 7
03-04-2020 (day 43): 11
03-05-2020 (day 44): 12
03-06-2020 (day 45): 14
 .
 .
 .
03-12-2020 (day 51): 40
03-13-2020 (day 52): 47
03-14-2020 (day 53): 54
03-15-2020 (day 54): 63
03-16-2020 (day 55): 85
03-17-2020 (day 56): 108
03-18-2020 (day 57): 118
```



```
Enter command> US
Population: 321,368,864
Life Expectancy: 79.68 years
Latest data:
 confirmed: 7,786
 deaths: 118
 recovered: 106
First confirmed case: 01-22-2020
First recorded death: 02-29-2020
Do you want to see a timeline? Enter c/d/r/n> r
Recovered:
02-09-2020 (day 19): 3
02-10-2020 (day 20): 3
02-11-2020 (day 21): 3
02-12-2020 (day 22): 3
02-13-2020 (day 23): 3
02-14-2020 (day 24): 3
02-15-2020 (day 25): 3
 .
 .
 .
03-12-2020 (day 51): 12
03-13-2020 (day 52): 12
03-14-2020 (day 53): 12
03-15-2020 (day 54): 12
03-16-2020 (day 55): 17
03-17-2020 (day 56): 17
03-18-2020 (day 57): 106
```

As mentioned earlier, use **getline**(cin, command) to obtain the user's input. This implies you should use this approach when getting additional input from the user, such as the "c/d/r/n" input that determines the type of timeline.

map

We used the **map** data structure in project #03. It stores (key, value) pairs in a red-black tree that is hidden from you. There are two ways to use map. The first is to search / insert using the **[ ]** operator, e.g.

```
map<keytype, valuetype> mymap;

mymap[key] = value;  // insert (key, value) in the map

cout << mymap[key] << endl;  // search by key for the value
```

This is very easy to use, with one caveat: if you search and the key is not in the map, it is inserted with a default value. So you generally want to use this approach for searching *only* when you know the key will be found. To search without inserting a default value, use the map's **find()** function, which returns an iterator (aka a pointer):

```
auto iter = mymap.find(key);  // search by key for the value

if (iter == mymap.end())  // not found:
{ … }
else  // found it!
{
  cout << iter->first << endl;   // this outputs the key
  cout << iter->second << endl;  // this outputs the value
}
```

We discussed map in class on days 10 and 11; lecture notes are available here.

Requirements

1. You must use a **map** where country name is the key; what the map contains for its value is up to you.
2. You can open an input file at most once; this applies to both daily reports and world facts files.
3. Your main.cpp program file must have a header comment with your name and a program overview. Each function must have a header comment above the function, explaining the function's purpose, parameters, and return value (if any). Inline comments should be supplied as appropriate.
4. You must have a single function for inputting a daily report, and call this function N times, once per daily report. Each command must be implemented using a function, and each function should be < 100 lines of code. This implies a complete solution must have at least 8 functions.
5. No global variables; use parameter passing and function return.
6. The **cyclomatic complexity** (CCN) of any function may not exceed 15, including main(). This will be reported when you submit on Gradescope, and you'll be warned if you exceed this threshold. In short, cyclomatic complexity is a representation of code complexity --- e.g. nesting of loops, nesting of if-statements, etc. You should strive to keep code as simple as possible, which generally means encapsulating complexity within functions instead of explicitly nesting code. Here's an example of simpler code with low CCN:

```
while (…)
{
   if (searchFunctionFindsWhatWeNeed(…))
     doSomething();

   next value;
}
```

   Here's an example of complex code with high CCN:

```
while (…)
{
   for (…)  // loop to do search
   {
      if (search condition is met)
      {
         for (…) // now do something:
         {
           …
         }
      }
   }

   next value;
}
```

   As a general principle, if we see code that has **more than 2 levels** of explicit looping --- an example of which is shown above --- we will score that code as 0, even if it's technically correct. The solution is to move one or more loops into a function, and call the function.

Grading, electronic submission, and Gradescope

Submission: "main.cpp" + additional .cpp, .h, and .hpp files that you submit

This is an extra credit project. To be eligible, you must have completed Project #05 (threaded AVL trees) by Sunday 3/22 (11:59pm) with an autograder score of 100. Then, if you successfully complete this project (i.e. score 100 on the autograder), then the final score on this project can be used to replace your lowest project score from projects 1-5. If it turns out your final score on this project is lower than your project scores on 1-5, then we'll use this score to replace your lowest 2 HW and lab scores. If it turns out your final score on this project is lower than all your HW and lab scores, it will be ignored. Do not ask to apply your score here to the midterm, it will not happen.

Your score on this project is based on two factors: (1) correctness as determined by your Gradescope submission, and (2) manual review by the TAs for commenting, style, and approach (e.g. use of functions and efficiency of solution). The entire project is worth 150 points: 100 points for correctness, and 50 points for commenting, style, and approach. The TAs will also review for adherence to requirements; breaking a requirement can result in a final score of 0 out of 150. We take all requirements seriously.

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

Policy

Late work is not accepted since this is an extra-credit project.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), 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 University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .