Homework 01

[2024-1] 데이터사이언스를 위한 컴퓨팅 1 (001)

Due: 2024 년 4 월 9 일 15 시 30 분

In this homework, you will implement CSV and JSON readers/writers using the C++ Standard Library.

CSV and JSON readers/writers are essential tools for data scientists. They provide functionality to efficiently read and write data, simplifying data analysis and processing tasks. Data scientists often use CSV or JSON formatted files to collect data from various sources, analyze it, and visualize it. These file formats make it easy to store and share data while preserving its structure.

In your implementation, you will extensively use data structures and fuctions in the C++ Standard Library. This will help you master various functionalities in this library, such as I/O streams and containers. Such experiences enhance C++ programming skills and provide the necessary techniques for data processing.

The goal of this homework is twofold:
(1) Learn to parse and write CSV/JSON data
(2) Learn to use various data structures and functions in the C++ Standard Library.

1. CSV Parser [50pts]

Implement CSV parsing functions that print the statistics of the data stored in a CSV (comma-separated values) file. Include your implementation in the `functions.cpp` file. You can modify other files (e.g., `main.cpp`) for your own use, but they will not be included in the grading process.

Instruction

The CSV file is structured as follows:

- The first line of the file serves as the header for column names, with subsequent lines containing data rows.
- All column names are unique.

(a) [10pts] Complete the `ParseCSV` function, which parses a CSV file and stores the column names and values in containers. The function takes three arguments: the input CSV filename, `header_cols`, and `data_map`. `ParseCSV` parses the contents of the CSV file to store them into the empty map `data_map`. Each key-value pair in `data_map` is constructed as [column name] – [column values] in the CSV file, with column values stored in a vector container. An

example below illustrates how `data_map` is constructed. Additionally, update the empty map `header_cols` with keys being the zero-based indices of the column names (i.e., 0, 1, 2, and so on) and values being the corresponding column names in the same order as in the CSV file. To test the `ParseCSV` function with `main.cpp`, you may execute it with 'a' as the first command-line argument and the input CSV filename as the second command-line argument.

[CSV file (sample.csv)]
```
StudentId,Level,Score
1234-56789,1,90
1234-54321,2,80
1234-54321,3,40
```

[Test with `main.cpp`]
```
./main a sample.csv
```

[`header_cols` is constructed as below]
{0: "StudentId", 1: "Level", 2: "Score"}

[`data_map` is constructed as below]
```
{"Level": {"1","2","3"},
"Score": {"90","80","40"},
"StudentId": {"1234-56789", "1234-54321", "1234-54321"}}
```

Please note that the input CSV file could include some dirty rows. Follow the rules below to deal with them:

- If the data row has a different number of elements than the header, ignore the row.
- If an element is enclosed by double quotes (") in the CSV file, any commas inside the double quotes are not considered as delimiters. When an element is enclosed by double quotes ("), the double quotes should also be kept in data_map.
- If any element of the data row is empty, ignore the row.

(b) [10 pts] Implement the `SumColumn` function, which sums all values in a target column. The function takes two arguments: the target column name and `data_map`. `data_map` is referenced from `data_map` constructed in subproblem (a). `SumColumn` locates the target column from `data_map` and sums all values in the vector container corresponding to the target column. Assume that the target column name is always one of the column names in the CSV file and the target column has only integer values. To test the `SumColumn` function with `main.cpp`, you may execute it with 'b' as the first command-line argument, the input CSV filename as the second command-line argument and target column name as the third command-line argument.

[CSV file (sample.csv)]

```
StudentId,Level,Score
1234-56789,1,90
1234-54321,2,80
1234-54321,3,40
```

[Test with `main.cpp`]

```
./main b sample.csv Score
```

```
SumColumn("Score", data_map)
>>> 210
```

(c) [15 pts] Implement the `FilterMostFrequentAvg` function, which calculates the average value for the most frequent key in a data map. The function takes three arguments: the key column name, the value column name, and `data_map`. `data_map` is referenced from the `data_map` constructed in subproblem (a). `FilterMostFrequentAvg` finds the most frequently appearing value in the key column. Then, the corresponding values in the value column are averaged and returned. Assume there is always a unique key value that appears most frequently, and the value column has only integer values. Key/Value columns always exist in the CSV file. To test the `FilterMostFrequentAvg` function with `main.cpp`, you may execute it with 'c' as the first command-line argument, the input CSV filename as the second command-line argument, and key and value column names as the third and fourth command-line arguments.

[CSV file (sample.csv)]

```
StudentId,Level,Score
1234-56789,1,90
1234-56789,3,20
1234-56789,3,40
1234-54321,2,80
1234-54321,3,40
```

[Test with `main.cpp`]

```
./main c sample.csv StudentId Score
```

```
FilterMostFrequentAvg ("StudentId", "Score", data_map)
>>> 50
```

(d) [15 pts] Implement the `SumTwoColumns` function, which sums two columns row-wise. The function takes five arguments: column name 1, column name 2, output file name, `header_cols` and `data_map`. The output file name is the name of the CSV file that will be created by `SumTwoColumns`. `header_cols` and `data_map` are referenced from `header_cols` and `data_map` constructed in subproblem (a). The newly created CSV file has an additional column compared to the input CSV file. The new column is named "SumOf[value column name 1]And[value column name 2]", whose values are the row-wise sum of the values

from column name 1 and column name 2. The header of the output file consists of the original columns in their original order, followed by the new column. Column names 1 and 2 always exist in the input CSV file. To test the `SumTwoColumns` function with `main.cpp`, you may execute it with 'd' as the first command-line argument, the input CSV filename as the second command-line argument, column name 1 and column name 2 as the third and fourth command-line arguments, and output filename as the fifth command-line argument.

[CSV file (sample.csv)]

```
StudentId,Level,Score
1234-56789,1,90
1234-56789,3,20
1234-56789,3,40
1234-54321,2,80
1234-54321,3,40
```

[Test with `main.cpp`]

```
./main d sample.csv Level Score output.csv
```

```
SumTwoColumns("Level", "Score", "output.csv", header_cols, data_map)
```

[New CSV file "output.csv" is created]

```
StudentId,Level,Score,SumOfLevelAndScore
1234-56789,1,90,91
1234-56789,3,20,23
1234-56789,3,40,43
1234-54321,2,80,82
1234-54321,3,40,43
```

2. JSON Parser

Using the popular `jsoncpp` library, parse a JSON (JavaScript Object Notation) file. Follow the instructions below for the implementation.

JSON is a string whose format resembles JavaScript object literal format. Inside JSON, you can include what you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals. The example below illustrates a simple JSON file with three members.

```
{
       "id" : "abc",
       "birthday" : "2012-08-15",
       "favorite-categories" : [1,7,8,10]
}
```

The JSON file can also include nested hierarchy, but we do not consider it in this homework. Please include your implementation in the functions.cpp file. You can modify other files (e.g., `main.cpp`) for your own use, but they will not be included in the grading process.

Instruction

The input JSON file is assumed to NOT have any nested hierarchy. In addition, only two types are used in the value part of each key-value pair: String or Array of Integers.

(a) Complete `ParseJSON` function, which parses a JSON file and stores the data in a structured container. The function takes two arguments: the input JSON filename and a reference to an empty `Json::Value` object. `ParseJSON` utilizes the `jsoncpp` library to parse the contents of the file and stores them in the input `Json::Value` object. This implementation is not graded but will serve as a foundation for the subsequent subproblems. To test `ParseJSON` function with `main.cpp`, you may execute it with 'a' as the first command-line argument and the input JSON filename as the second command-line argument.

[JSON File (sample.json)]

```
{
       "id" : "abc",
       "birthday" : "2012-08-15",
       "favorite-categories" : [1,7,8,10]
}
```

[Test with `main.cpp`]

```
./main a sample.json
```

```
A new Json::Value object is created with the key-value pairs in the input
JSON file as its contents.
```

(b) [15 pts] Implement the `CheckTarget` function, which checks whether a target key exists in a `Json::Value` object. The function takes two arguments: a `Json::Value` object constructed from (a) and a target key. If the target key exists in the JSON object, `CheckTarget` returns a `std::unique_ptr` that points to the value of the target key as a `string`. Otherwise, `CheckTarget` returns `nullptr`. Assume that if the target key exists in the JSON file, its value is always a string. To test `CheckTarget` function with `main.cpp`, you may execute it with 'b' as the first command-line argument, the input JSON filename as the second command-line argument and target key as the third command-line argument.

[JSON File]

```
{
       "id" : "abc",
       "birthday" : "2012-08-15",
       "favorite-categories" : [1,7,8,10]
}
```

[Test with `main.cpp`]

```
./main b sample.json id
```

[root is a `JSON::Value` object constructed in (a)]

```
*CheckTarget(root, "id")
>>> "abc"
CheckTarget(root, "password")
>>> nullptr
```

(c) [15 pts] Implement the `CompleteList` function, which iterates through all fields of a `Json::Value` object and collects string values. The function takes two arguments: the `Json::Value` object constructed from (a) and an empty list container. All values of string type are inserted into the input list container in descending order of their keys. (Note: In `Json::Value`, keys are always stored in ascending order.) When there is no member whose value is a string type, insert "NO_STRING" into the input list.To test the `CompleteList` function with `main.cpp`, you may execute it with 'c' as the first command-line argument and the input JSON filename as the second command-line argument.

[JSON File]

```
{
       "id" : "abc",
       "birthday" : "2012-08-15",
       "favorite-categories" : [1,7,8,10]
}
```

[Test with `main.cpp`]

```
./main c sample.json
```

[root is a `JSON::Value` object constructed in (a)]

```
std::list<std::string> lst;
CompleteList(root, lst);
lst
>>> list({"abc","2012-08-15"})
```

(d) [20 pts] Implement the `SumTwoMembers` function, which adds two fields element-wise. This function takes four arguments: key name 1, key name 2, output file name, and a `Json::Value` object constructed from (a). Key names 1 and 2 are among the members in the input JSON file. Their values are assumed to be integer arrays. The output file name is the name of the JSON file that will be created by SumTwoMembers. The newly created JSON file has an additional field named "SumOf[key name 1]And[key name 2]", and its value is an array of the element-wise sum of the two fields corresponding to key name 1 and key name 2. Other than this new field, the output JSON file should have the same contents as the input JSON file. When the two integer arrays to be summed have different lengths or are both empty, the new member is not added. In other words, the output JSON file will have the same contents as the input JSON file in this case. To test `SumTwoMembers` function with `main.cpp`, you may execute it with 'd' as the first command-line argument, the input JSON filename as the second command-line argument, key names 1 and 2 as the third and fourth command-line arguments and output filename as the fifth command-line argument.

[JSON File]

```
{
       "id" : "abc",
       "birthday" : "2012-08-15",
       "favorite-categories1" : [1,7,8,10],
       "favorite-categories2" : [3,5,7,12]
}
```

[Test with `main.cpp`]

```
./main d sample.json favorite-categories1 favorite-categories2 output.json
```

```
SumTwoMembers("favorite-categories1","favorite-categories2",
"output.json", root)
```

[New JSON file "output.json" is created]

```
{
   "SumOffavorite-categories1Andfavorite-categories2" : [ 4, 12, 15, 22 ],
   "birthday" : "2012-08-15",
   "favorite-categories1" : [ 1, 7, 8, 10 ],
   "favorite-categories2" : [ 3, 5, 7, 12 ],
   "id" : "abc"
}
```

3. Submission Instructions

- For each problem, write your functions in the functions.cpp file prepared for that problem, and submit this file to Gradescope for grading. You can modify other files for your own testing, but keep in mind that **only functions.cpp will be graded using the original header files and main.cpp.**
- Grading will be conducted using the C++11 standard. Our compilation process is as follows:
  - `$ g++ -c main.cpp -o main.o -std=c++11` (using the original main.cpp)
  - `$ g++ -c functions.cpp -o functions.o -std=c++11` (using your functions.cpp)
  - `$ g++ main.o functions.cpp -o main`
- When submitting your codes, ensure that all test codes printing out `stdout` are removed from `functions.cpp`, unless explicitly instructed otherwise. Grading relies solely on the `stdout`, and including any unrelated codes may result in inaccurate grading.
- You should NOT share your code with other students. Any student found to have a high level of code similarity, as determined by our similarity detection process, may face severe penalties.
- If you submit late, grace days will be automatically deducted. You can utilize 5 grace days throughout the semester for homework submissions. Grace days are counted based on 24-hour increments from the original due time. For instance, if an assignment is due on 4/9 at 3:30pm, submitting it 30 minutes late will count as using one grace day, while submitting it on 4/10 at 9pm will count as using two grace days. Late submissions after exhausting all grace days will NOT be accepted.
- For questions about this homework, use the “Homework 1” discussion forum on eTL.
- Failing to adhere to the submission guidelines may result in penalties applied without exception.