Reader Writer of Modern C++ Objects To/From a JSON-like Store

Project Winter 2015

March 24, 2015

Requirements

Using **Modern C++ write code** for the following:

1. The two template functions declared in `src/project.h`.
2. The two classes: `cartoon` and `course` that derive from `object` (`src/object.h`). Some of `object` implementation in `cartoon` and `course` is mandated through overrides for the two pure virtual functions: `to_dsv()` and `to_json()` in `src/object.h`. In addition, `src/main.cpp` also needs getter member functions to retrieve member data of `cartoon` and `course` objects. Member data for `cartoon` and `course` can be gleaned from the JSON-like files in the `data` directory, for example: `cartoon` will have: type, name, and likes as it's data-members and when reading `data/03-one_cartoon_object.json` it will populate those data-members with: cat, Tom, and mice.
3. *Optionally*, you may create and write code for any additional classes or helper functions that you need to compile and run the given `src/main.cpp` correctly.
4. In your code comments, state which workshop learning objectives you've used in your project solution. You may also add comments to `src/main.cpp` to indicate which learning objectives `src/main.cpp` uses from the workshops, however, when modifying `src/main.cpp` only add comments, do not add, remove, or change any of the existing C++ statements otherwise your project submission will be rejected.

Testing Project Requirements

The `src/main.cpp` tests the following situations [1]:

- Read a JSON-like [2] file and create a `std::vector` of objects found in that file, for example: when template function `readFromJsonLike()` in `src/project.h` reads `data/05-array_with_few_cartoon_objects.json`, `src/main.cpp` verifies that objects returned are of type `cartoon` and available as a comma-separated-value string. **NOTE:** Both `cartoon` and `course` derive from `object`.
- Filter a returned collection based on some search criteria and save that to a JSON-like file, for example: using the same `data/05-array_with_few_cartoon_objects.json` file, `src/main.cpp` will test that no dog named Spike exists (it doesn't) from the returned filtered collection and save the (empty) results to `data/got_array_with_no_cartoon_object.json`
- Compare JSON-like data saved through a call to your implementation of template function `writeToJsonLike()` in `src/project.h` with the expected data and format, for example: file `data/got_array_with_no_cartoon_object.json` will be compared, line by line, with `data/expected_array_with_no_cartoon_object.json`. **NOTE:** Output saved by your solution must be valid according to the online JSON validator, alternatively you can look at what the expected output should be by examining: `data/expected_*.json`.

[1] `correct_output.txt` has results of a successful run when compiled with `src/main.cpp` and tested with JSON-like files from `data/*`
[2] JSON-like format is a subset of JavaScript Object Notation and is discussed in detail later in this document.

In addition, to the above tests, the project tests the following boundary conditions for JSON-like data files:

- reading from JSON-like files containing zero, one, few, and many objects
- writing to JSON-like files containing zero, one, and many objects
- two different types of JSON-like data files (`*cartoon*.json` and `*course*.json`) will be used to test that two different `std::vector` of those objects are returned. So `data/*_cartoon*.json` should return a `stl::vector` containing `cartoon` objects, and `data/*_course*.json` should return a `stl::vector` containing `course` objects.

Project Purpose

The project uses learning objectives discussed and used in the weekly workshops to help solve a real-world problem involving persistence of C++ objects in a commonly accepted format. In detail the purpose of the project is as follows:

- To build and test Modern C++ code that solves a real world problem.
- Ensure the solution runs on Windows and on Unix (matrix).
- Show how learning objectives from the workshops can be grouped together to ease solution of a relatively complex programming problem by using Modern C++ techniques learned to handle the following project tasks:
  - Read from and write to a JSON-like file line-by-line
  - Split a JSON-like line into constituent tokens and create C++ objects from them
  - Create, store, and retrieve a type-safe collection of objects derived from `src/object.h`
- Understand how inheritance and composition can work together to reduce complexity and increase reusability.

HOWTO solve

First ensure you know and understand the learning objectives taught in the workshops, so for example: a reference implementation of the project specs used the following workshop learning objectives (a full list of all workshop learning objectives are at the end of this file):

```
- [w01] to guard a class definition from repetition
- [w01] to define a class within its own namespace

- [w02] work with objects of the string class
- [w02] declare and use enumeration constants
- [w02] retrieve data from and backup data to a text file

- [w03] retrieve data from a text file using an ifstream object

- [w04] design and code a class template

- [w05] design and code a composition of objects
- [w05] read records from a file into a string object
- [w05] parse a string object into components based on simple rules

- [w06] use a lambda expression to define an operation on a set of data values
- [w06] throw and report an exception

- [w07] store and manage polymorphic objects using an STL vector
- [w07] report and handle an exception

- [w08] copy data from a file into a sequential container
- [w08] operate identically on each value in a STL collection using a lambda expression and
        accumulate results of the operation library of the STL

- [w09] use smart pointers
```

```
The following learning objectives were slightly changed from their original statements:
  - [w04] store key-value information
  - [w08] collect and use data values using the algorithm library of the STL
```

Before you start on the problem, you will need to understand how to run and test your code within the given build tool. This section explains the build process in further detail. The directory layout is as follows:

```
.
|-- bin/
|-- build
|-- build.bat
|-- correct_output.txt
|-- data/
|-- mk/
|-- README.txt
`-- src/
```

The purpose of each directory is as follows:

- `bin` contains the project executables for Windows (`project.exe`) and Unix (`project.out`)
- `data` contains the sample data files needed by the project
- `mk` contains the make files needed by the project build scripts
- `src` contains the project source files
- the top-level directory contains: `README.txt`, build files for Windows (`build.bat`) and Unix (`build`), and a reference of what is valid output (`correct_output.txt`) for the given `src/main.cpp`

While building your project, put your project source files in `src`, follow instructions in `README.txt` on HOWTO build the project on Windows and on Unix (matrix), and run `build` (on matrix) and `build.bat` (on Windows) from the top-level directory (the directory containing `README.txt`)
In case you make compiler or runtime mistakes, it is best to do the following:

- On Windows, run the build command as follows

  ```
  build 2>&1 >err
  ```

- On matrix, run the build command as follows

  ```
  ./build 2>err
  ```

A suggested way of approaching the problem is to comment out the main in stages, build, compile and test. Once you get the expected output move onto the next part of the project. A linear sequence of steps would be:

1. Get your project to work with `data/*empty*.json` files. These files contain no JSON-like objects. The following edge cases for empty files are tested:
   - An empty object (`00-empty_object.json`)
   - An empty array (`01-empty_array.json`)
   - An empty array with an empty object (`02-empty_array_with_empty_object.json`)
2. Next get your project to work with `data/*one_cartoon-object.json`. These files contain just one object, which can either be by itself in the JSON-like file or inside an array. Sample files for one JSON-like object files are:
   - File with only one object (`03-one_cartoon_object.json`)
   - An array with only one object (`04-array_with_one_cartoon_object.json`)
3. Handle JSON-like files containing more than one `cartoon` and `course` objects. See sample files:
   - An array with few objects (`05-array_with_few_cartoon_objects.json` and `06-array_with_few_course_objects.json`)
   - An array with many objects (`07-array_with_many_course_objects.json`)

4. Once you have finished reading the files correctly, handle the writing of objects to JSON-like format. Again, these are graduated in terms of complexity with saving of an empty `cartoon` list to `got_array_with_no_cartoon_object.json`, saving of list containing one `course` object to `got_array_with_one_bsd_object.json`, and saving of list containing many `course` objects to `got_array_with_many_bsd_objects.json`. The format of the saved files will be compared with their `expected_array_with_*.json` versions. So writing:

   - an empty json file, `got_array_with_no_cartoon_object.json`, should be formatted exactly like `expected_array_with_no_cartoon_object.json`
   - a solitary object in an array, `got_array_with_one_bsd_object.json`, should be formatted exactly like `expected_array_with_one_bsd_object.json`
   - a collection of objects in an array, `got_array_with_many_bsd_objects.json`, should be formatted exactly like `expected_array_with_many_cpa_objects.json`

5. Redirect your final correct output to file and compare with the given `correct_output.txt`. If there are no differences (**NOTE:** `correct_output.txt` was created on Windows) between your output and `correct_output.txt` then your project is working according to specs (on Windows).

6. Retest the code on matrix and verify that there are no compiler errors or warnings and that the output is the same as on Windows.

7. Follow your professor's instructions on HOWTO hand-in the project.

Simply speaking, when your build output matches the expected build output given in the `correct_output.txt`, you are done.

Workshop Learning Objectives

This is the complete list of all workshop learning objectives.

```
- [w01] to guard a class definition from repetition
- [w01] to define a class within its own namespace
- [w01] to declare a local variable that lasts the lifetime of the program
- [w01] to link to a variable in another translation unit
- [w01] to pass arguments to program from the command line

- [w02] work with objects of the string class
- [w02] declare and use enumeration constants
- [w02] retrieve data from and backup data to a text file
- [w02] use the correct constant type in initializations
- [w02] move data between unsigned and signed integers

- [w03] retrieve data from a text file using an ifstream object
- [w03] implement copy semantics for a class with a resource
- [w03] implement move semantics for a class with a resource
- [w03] identify the processing-intensive operations in copy and move assignments
- [w03] reflect on the material learned through this workshop

- [w04] design and code a class template
- [w04] store key-value information in a pair of parallel arrays

- [w05] design and code a composition of objects
- [w05] read records from a file into a string object
- [w05] parse a string object into components based on simple rules

- [w06] use a lambda expression to define an operation on a set of data values
- [w06] prevent the copying, moving and assigning of an object
- [w06] throw and report an exception

- [w07] store and manage polymorphic objects using an STL vector
- [w07] report and handle an exception
- [w07] store a set of uniform rates for all instances of a class using a class array

- [w08] copy data from a file into a sequential container
- [w08] sort the data values in a data set using the algorithm library of the STL
- [w08] accumulate data values using the numeric library of the STL
- [w08] operate identically on each value in a data set using a lambda expression and
        accumulate the results of the operations using the numeric library of the STL

- [w09] create a program component of quadratic complexity
- [w09] use a smart pointer to move an object
- [w09] reflect on the topics learned in this workshop

- [w10] execute partitioned data on two or more threads
- [w10] write a set of characters to a file in binary mode
- [w10] read a set of characters from a file in binary mode
- [w10] bind a function to its arguments


- [w04] reflect on the material learned in this workshop
- [w05] reflect on the material learned in this workshop
- [w06] reflect on what you have learned in this workshop
- [w07] reflect on what you have learned in this workshop
- [w10] reflect on the experience gained in this workshop
```

Grammar of JSON-like spec

IMPORTANT: JSON-like files used by this project, must be valid and formatted according to the online JSON validator as the output from that site was used by our project (the online validator, http://jsonlint.com, indents our valid JSON-like using spaces and newlines). Definitions and rules for our JSON-like grammar are as follows:

- A JSON-like file must have either one array or one object
- An array is defined by []. It can be empty or contain: one or more objects. Objects are separated by comma (,)
- An object is defined by {}. It can be empty or contain: one or more elements. Elements are separated by comma (,)
- An element is a name-value pair where name and value are separated by a colon (:)
- name must be a non-empty string enclosed by quotation ("")
- value must be a string (empty string values are allowed) enclosed by quotation ("")

A formal specification of the above rule set is:

```
json-like   :  object
            |  array
            ;

object      :  '{' '}'
            |  '{'  members  '}'
            ;

members     :  pair
            |  pair ',' members
            ;

pair        :  STRING ':' STRING
            ;

array       :  '[' ']'
            |  '[' elements ']'
            ;

elements    :  object
            |  object ',' elements
            ;
```