JSON Tutorial

The *JavaScript Object Notation* (or JSON) data serialization format, and many like it[1], are frequently used in industry, and *every* computer science student should be exposed to it. These types of formats are frequently used for web requests and to pass data from one programming environment to another (e.g. Objective-C to Swift). We will be using JSON for every programming project during the quarter for Input / Output of your programs and for the grading of your programs.

The running example of this document is illustrated below. In the first project, we are writing code to examine sorting algorithms and the data they produce. JSON files for this program will be formatted like this[2]:

```
{
  "Sample1": [-319106570,811700988,1350081101,1602979228],
  "Sample2": [-319106570,811700988,797039,-1680733532],
  "metadata": {
    "arraySize":4,
    "numSamples":2
  }
}
```

Data input to your program will come in the format of a JSON file that has one or more *samples* (an array to be sorted) and metadata (e.g. # of samples, size of the arrays) about the samples.
This file consists of two arrays of integers (samples) to be sorted and metadata about these samples: there are two of them and each is of size 4.

Key / Value Pairs

A JSON object consists of a collection of *key* / *value* pairs. It turns out you are already familiar with the concept of key / value pairs: an array can be viewed as

[1] An incomplete list includes BSON, YAML, Protocol Buffers, MessagePack, XML, etc. We won't consider other data serialization formats in this course, but if you are interested, see `https://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats`.

[2] This file is `SampleExample.json` from `Program1Files.zip`.

a collection of key / value pairs where the keys are in the range $0, 1, \ldots, n-1$ for an array with n elements. For example, the array $A = [-3, 9, 4]$ has $A[1] = 9$, and here the *key* is 1 and the *value* is 9. For an array we call a key an *index* and it is simply an integer. For JSON objects, keys will be strings, and unlike arrays, values can take on differing types. The type of a key may be a string, a number, a boolean, an array, or a JSON object.

Consider again the previous example:

```
{
  "Sample1": [-319106570,811700988,1350081101,1602979228],
  "Sample2": [-319106570,811700988,797039,-1680733532],
  "metadata": {
    "arraySize": 4,
    "numSamples": 2
  }
}
```

This object has three keys: `Sample1`, `Sample2`, and `metadata`. The value of `Sample1` is an array of 4 integers, $[-319106570, 811700988, 1350081101, 1602979228]$. The value of `sample2` is also an array of 4 integers, $[-319106570, 811700988, 797039, -1680733532]$. The value of the `metadata` key is a JSON object itself:

```
{
  "arraySize": 4,
  "numSamples": 2
}
```

This object has two keys; the value of `arraySize` is 4 and `numSamples` is 2.

How to examine a JSON file

Since JSON files are often used for web requests, they often come in a compact form that omits whitespace. Because of this, it can be difficult to eyeball the contents of a JSON file, as the example above would be stored like this:

```
{"Sample1":[-319106570,811700988,1350081101,1602979228],"Sample2":[
-319106570,811700988,797039,-1680733532],"metadata":{"arraySize": 4,
"numSamples": 2}}
```

For readability, I've had to format the above string on three lines, but in a JSON file this would be a single line.

In order to examine the contents of the JSON object represented by this string, copy the string, and then paste it into a JSON viewer (for example, use `https://jsonformatter.curiousconcept.com/` and click "Process"). Doing so results in the following more readable format:

```
{
   "Sample1":[
      -319106570,
      811700988,
      1350081101,
      1602979228
   ],
   "Sample2":[
      -319106570,
      811700988,
      797039,
      -1680733532
   ],
   "metadata":{
      "arraySize":4,
      "numSamples":2
   }
}
```

Still, it can be difficult to read JSON files in this format when the data is large (e.g. thousands of entries).

JSON for Modern C++

Reading (or *parsing*) a JSON object from a JSON file is a difficult task: do not write your own code for this. The programming projects require you to use the 3rd-party library https://github.com/nlohmann/json. This is done by adding `json.hpp`[3] to your project directory and adding `include "json.hpp"` to your source code. The github repo at `https://github.com/nlohmann/json` contains documentation, but it is only useful if you are comfortable with the STL, iterators, and so forth. As such, I have compiled a list of operations that you will need to be familiar with.

Reading a JSON file

Use `ifstream`[4] and the stream operator to open a JSON file. The steps are:

1. Open a file using an `ifstream` object.
2. Declare a `nlohmann::json` object.
3. Check if the file is correctly opened.

[3] This file is over 14,000 lines of code. It's a good experience to open it and take a look, but don't waste project time examining it.

[4] Visit `https://www.cprogramming.com/tutorial/lesson10.html` if you are unfamiliar or rusty with C++.

4. Parse the file with the >> operator.

As an example:

```
std::ifstream file;
file.open(filename);
nlohmann::json jsonObject;
if (file.is_open()) {
  file >> jsonObject;
}
```

As a first step[5], you should write a simple program from scratch that will read in `SampleExample.json` and print out its contents. You should be able to accomplish this in 12-15 lines of code. There are two ways to print a json object. First, you can use `std::cout` to print output on a single line, like this:

```
std::cout << jsonObject;
```

Putting this all together, if we use:

```
std::ifstream file;
file.open("SampleExample.json");
nlohmann::json jsonObject;
if (file.is_open()) {
  file >> jsonObject;
}
std::cout << jsonObject;
```

The output would be the following (I've formatted this on two lines, but this will print a single line):

```
{"Sample1":[-319106570,811700988,1350081101,1602979228],"Sample2": [-319106570,
811700988,797039,-1680733532],"metadata": {"arraySize": 4,"numSamples": 2}}
```

The second option for printing is to use the `dump` function:

```
std::ifstream file;
file.open("SampleExample.json");
nlohmann::json jsonObject;
if (file.is_open()) {
  file >> jsonObject;
}
std::cout << jsonObject.dump(2) << "\n";
```

The parameter "2" which gives human readable output:

[5]When learning how to use a 3rd party library or any new programming language, start *slowly* and build up to larger tasks.

```
{
  "Sample1": [
    -319106570,
    811700988,
    1350081101,
    1602979228
  ],
  "Sample2": [
    -319106570,
    811700988,
    797039,
    -1680733532
  ],
  "metadata": {
    "arraySize": 4,
    "numSamples": 2
  }
}
```

Accessing Values

To access a value associated with a key, the simplest method is to use square braces (the way you index an array) along with the key. For example, if we read the file `SampleExample.json` to `jsonObject`, then `jsonObject` now has values represented by:

```
{
  "Sample1": [-319106570,811700988,1350081101,1602979228],
  "Sample2": [-319106570,811700988,797039,-1680733532],
  "metadata": {
    "arraySize": 4,
    "numSamples": 2
  }
}
```

The code

```
int n = jsonObject["metadata"]["arraySize"];
```

sets the value of `n` to 4. We can also set values this way. Using

```
jsonObject["metadata"]["filename"] = "SampleExample.json";
```

results in the JSON object represented by:

```
{
  "Sample1": [-319106570,811700988,1350081101,1602979228],
  "Sample2": [-319106570,811700988,797039,-1680733532],
  "metadata": {
```

```
    "arraySize": 4,
    "filename": "SampleExample.json",
    "numSamples": 2
  }
}
```

We also access JSON objects nested in JSON objects this way. That is, the code:

```
nlohmann::json metadataObj = jsonObject["metadata"];
```

will set the variable `metadataObj` to the the JSON object represented by the string

```
{
  "arraySize": 4,
  "filename": "SampleExample.json",
  "numSamples": 2
}
```

Now `metadataObj["arraySize"]` is 4, and so on.

Care must be taken as it is possible to request values for keys that do not exist. Expanding on the code above, consider the following code.

```
std::ifstream file;
file.open("SampleExample.json");
nlohmann::json jsonObject;
if (file.is_open()) {
  file >> jsonObject;
}
int n = jsonObject["someKey"];
```

There is no key `someKey` in this JSON object, so the JSON library throws a runtime error that may look like this:

```
libc++abi.dylib: terminating with uncaught exception of type
nlohmann::detail::type_error: [json.exception.type_error.302] type must be
number, but is null
Abort trap: 6
```

Note that this can not be caught at compile time because the keys in a JSON object are dynamic. This exception can be handled, but I am lucky to see this on my machine because trying to access non-existent keys with brackets causes undefined behavior in general. Note that there is no meaningful information in regards to *where* the error is coming from. You can obtain this by using `at` and a `try/catch` block, like this:

```
try {
  int n = jsonObj.at("someKey");
} catch (nlohmann::json::out_of_range& e) {
  std::cout << e.what() << '\n';
}
```

which outputs the following instead:

```
[json.exception.out_of_range.403] key 'someKey' not found
```

This doesn't actually terminate the program unless you explicitly call `exit` in your `try/catch` block. You can get even more meaningful information if you couple this approach with the macros `__PRETTY_FUNCTION__` and `__LINE__`, like this (I've included line numbers for clarity):

```
12: try {
13:   int n = jsonObj.at("someKey");
14: } catch (nlohmann::json::out_of_range& e) {
15:   std::cout << __PRETTY_FUNCTION__ << ":" << __LINE__ << ":"
16:             << e.what() << '\n';
17: }
18: std::cout << __PRETTY_FUNCTION__ << ":" << __LINE__ << std::endl;
```

which gives output:

```
int main():15:[json.exception.out_of_range.403] key 'someKey' not found
int main():18
```

The macro `__PRETTY_FUNCTION__` expands to a human-readable function signature for the function this macro is placed in and the macro `__LINE__` expands to the line number for the line this macro is placed on.

Iterating over Keys

You can iterate over all key / value pairs at the *top level* by using the following code snippet:

```
for (auto itr = jsonObject.begin(); itr != jsonObject.end(); ++itr) {
  std::cout << "key: " << itr.key() << " value: " << itr.value() << '\n';
}
```

The *iterator* `itr` starts by pointing to the first key / value pair (i.e. `jsonObject.begin()`), increments to the next key / value pair using `++itr`, and stops when it is *past* the last key / value pair (i.e. `jsonObject.end()`[6].) The `auto` keyword is a lazy way to declare the type of `itr`. The code `itr.key()` returns the key of the key / value pair that `itr` points to, and `itr.value()` returns the value of the key / value pair that `itr` points to.
The above code would print:

[6] Note that `jsonObject.end()` is *past* the last key / value pair, it is not the last key itself. This is the standard behavior of STL containers.

```
key: Sample1 value: [-319106570,811700988,1350081101,1602979228]
key: Sample2 value: [-319106570,811700988,797039,-1680733532]
key: metadata value: {"arraySize":4,"numSamples":2}
```

Note that the keys are being printed in `lexicographic order`, and that the keys of `metadata`'s value are not iterated over. That is, the above code does not iterate over the JSON object:

```
{
  "arraySize": 4,
  "numSamples": 2
}
```

Instead, it only iterates over keys at the *top level*.

When the key is `Sample1`, the value is a JSON object representing an array of integers that you can iterate over like this:

```
for (auto arrayItr = jsonObect["Sample1"].begin();
     arrayItr != jsonObject["Sample1"].end();
     ++arrayItr) {
  std::cout << *arrayItr << " ";
}
```

Unlike our previous example, `arrayItr` only points to *values* since the keys are simply indices. To retrieve the value, we use the star operator (e.g. `*arrayItr`). This would print:

```
-319106570 811700988 1350081101 1602979228
```

This time the elements are printed in the order they are listed in the array, as opposed to lexicographic order.