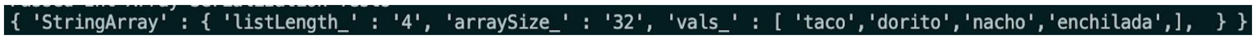
Serialization Memo

Design:

For our serialization library we focused on a readable implementation as opposed to a more efficient byte stream representation. This was due to a multitude of reasons, but primarily for readability during debugging in its future use cases. We believe that when it comes time to use our Serialization api, it will be in a context where any insight we can have into the data being sent/received throughout its lifecycle will be of great value. Whenever we've worked with data sent or received from a server structure in the past we've found it incredibly useful to be able to parse exactly what is going on at each step - especially when it comes to the deserialization. While a byte stream might be slightly easier to implement, it's harder to decipher where bogus data and bugs could be coming from if you can't easily read what data is being deciphered. We hope that the extra design time to create our readable format will pay off in avoiding these future debugging headaches down the line.

To illustrate the implementation of our approach to serialization, the bundled main.cpp provides examples of serializing the typed arrays from our dataframe implementation, and the serialization on prototypes of the classes involved in the server/client architecture These are provided in a header file called "network.h"

For inspiration, we looked to JSON - and created a rudimentary implementation of the popular data framework in CwC with output that looks like this:



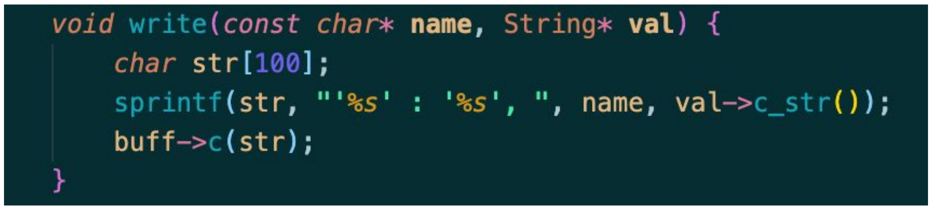
```
{ 'StringArray' : { 'listLength_' : '4', 'arraySize_' : '32', 'vals_' : [ 'taco','dorito','nacho','enchilada',],  } }
```

Putting the name of the class first, to identify each object implementation followed by all the specific object fields in an easy to interpret format. For now, we're less concerned with a completely identical to JSON implementation (you'll notice a few trailing commas) but should this be a library we build out further - it's on the top of our list. Since the deserialization keys off of just brackets, braces, and quotes, the commas are there solely for human readability anyway.

Serial.h

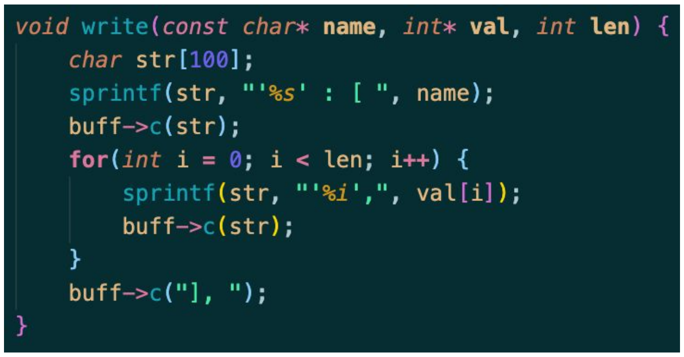
- **char* get()** - gets the current char* representation of the StrBuff holding the serialization
- **void write(const char* name, value val)** - many different implementations for all necessary data-types. Adds a keyed value to the serialization buffer
- **void write(const char* name, value val, int length)** - specific implementation for adding arrays, needs length as well
- **void initSerialize(const char* name)** - called before write methods to set up the JSON object with the className
- **void endSerialize()** - called after all writes are complete to close necessary brackets

The Serializable class has a StrBuff field, and multiple implementations of "write" that support all the necessary data types to be added to the eventual Serialization with a provided 'name' to key off of in deserialization.



```
void write(const char* name, String* val) {
    char str[100];
    sprintf(str, "'%s' : '%s', ", name, val->c_str());
    buff->c(str);
}
```

String implementation*

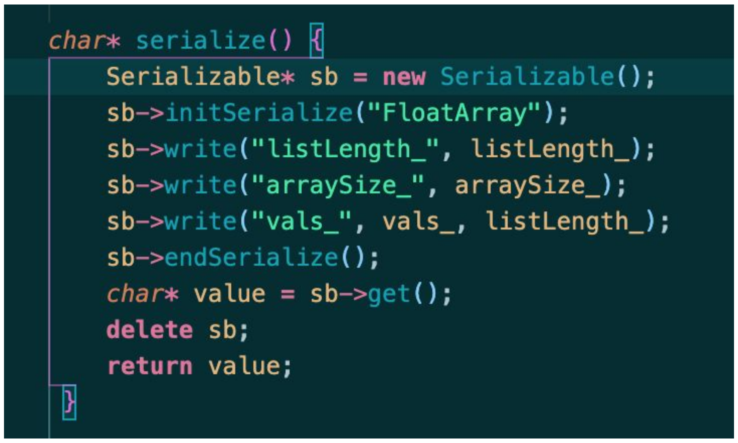


```
void write(const char* name, int* val, int len) {
    char str[100];
    sprintf(str, "'%s' : [ ", name);
    buff->c(str);
    for(int i = 0; i < len; i++) {
        sprintf(str, "'%i',", val[i]);
        buff->c(str);
    }
    buff->c("], ");
}
```

Array implementation

The *name* would be the name you want to give that variable or object, with val being the object itself - in this case a String* or int*. Then, the name and val are paired JSON style and added to the StrBuff.

The use of these write methods allows our implementations of Serializable to look as clean as if we were using a byte stream serialization and therefore makes accessing this library very intuitive. Simply call write on the Serializable with the data name and the data itself for each piece you're trying to preserve. The only additional methods to keep in mind when using our JSON Serialization is calling "initSerialize" before providing the data through the write methods, and "endSerialize" once all the data has been added. These add the open and close brackets surrounding the class respectively.



```
char* serialize() {
    Serializable* sb = new Serializable();
    sb->initSerialize("FloatArray");
    sb->write("listLength_", listLength_);
    sb->write("arraySize_", arraySize_);
    sb->write("vals_", vals_, listLength_);
    sb->endSerialize();
    char* value = sb->get();
    delete sb;
    return value;
}
```

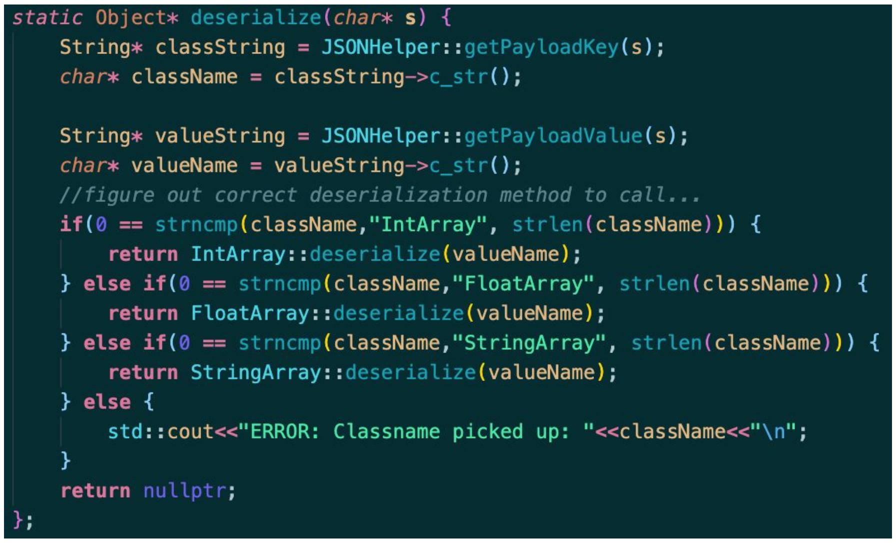
Serialize method for IntArray

Lastly, there is a "get" that returns the char* of the String* of StrBuff.

Deserialize.h

- **static Object* deserialize(char* c)** - given a serialized string, parses object type and calls the correct Class's deserialization method

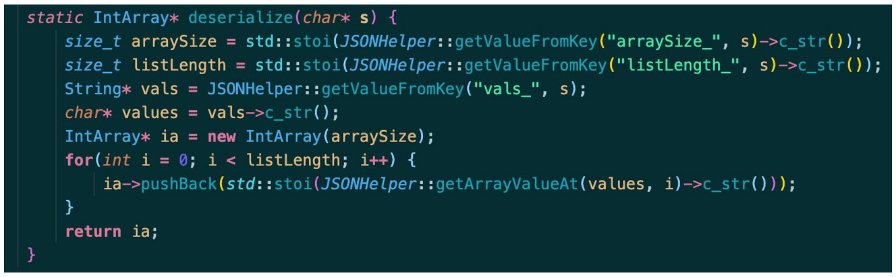
Deserializable is a helper class used to illustrate one style of deserialization that our implementation services with relative ease. When passed a serialized class, it can dispatch to the correct deserialization method based on the class tag that serves as the identifier in each of our JSON serializations.



```
static Object* deserialize(char* s) {
    String* classString = JSONHelper::getPayloadKey(s);
    char* className = classString->c_str();

    String* valueString = JSONHelper::getPayloadValue(s);
    char* valueName = valueString->c_str();
    //figure out correct deserialization method to call...
    if(0 == strncmp(className,"IntArray", strlen(className))) {
        return IntArray::deserialize(valueName);
    } else if(0 == strncmp(className,"FloatArray", strlen(className))) {
        return FloatArray::deserialize(valueName);
    } else if(0 == strncmp(className,"StringArray", strlen(className))) {
        return StringArray::deserialize(valueName);
    } else {
        std::cout<<"ERROR: Classname picked up: "<<className<<"\n";
    }
    return nullptr;
};
```

Choosing correct deserialize method based on class



```
static IntArray* deserialize(char* s) {
    size_t arraySize = std::stoi(JSONHelper::getValueFromKey("arraySize_", s)->c_str());
    size_t listLength = std::stoi(JSONHelper::getValueFromKey("listLength_", s)->c_str());
    String* vals = JSONHelper::getValueFromKey("vals_", s);
    char* values = vals->c_str();
    IntArray* ia = new IntArray(arraySize);
    for(int i = 0; i < listLength; i++) {
        ia->pushBack(std::stoi(JSONHelper::getArrayValueAt(values, i)->c_str()));
    }
    return ia;
}
```

Example deserialize method

Seen here implemented with the array classes, deserialize takes in any JSON representation and applies the correct deserialization meaning you don't necessarily need to know the type on initial receiving of the data.

You'll also notice a couple method calls "getPayLoadKey" and "getPayloadValue" from a class called JSONHelper, but we'll get to that in a minute.

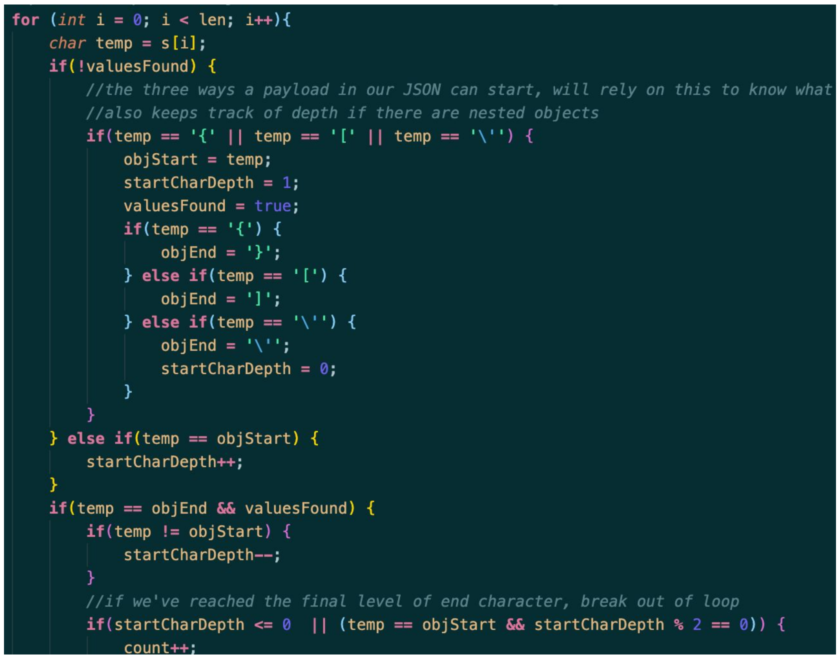
jsonHelper.h

- **static String* getPayloadKey(char* c)** - takes a complete serialized object and returns the className
- **static String* getPayloadValue(char* c)** - takes a complete serialized object and returns the value for the className returned by 'getPayloadKey'
- **static String* getValueFromKey(char* name, char* s)** - given a key (name) and a serialized string, returns the value for that key

- **static int arrayLen(char* s)** - given a serialized array, returns how many entries are in it
- **static String* getArrayValueAt(char* s, int index)** - given a serialized array, returns the entry at index i

jsonHelper is a bunch of specialty functions to deal with accessing data from a JSON serialization. This includes getting the Object name, getting the Object value, getting a value given a key, the length of encoded JSON arrays, and getting an individual value at the index of a JSON array.

The primary difficulties in handling JSON code comes from the multiple types of symbols data can be wrapped in: mainly " { } ", " [ ] ", and " ' " . Therefore, each needs to be handled specially or stacked in the case of nested punctuation.



```
for (int i = 0; i < len; i++){
    char temp = s[i];
    if(!valuesFound) {
        //the three ways a payload in our JSON can start, will rely on this to know what
        //also keeps track of depth if there are nested objects
        if(temp == '{' || temp == '[' || temp == '\'') {
            objStart = temp;
            startCharDepth = 1;
            valuesFound = true;
            if(temp == '{') {
                objEnd = '}';
            } else if(temp == '[') {
                objEnd = ']';
            } else if(temp == '\'') {
                objEnd = '\'';
                startCharDepth = 0;
            }
        }
    } else if(temp == objStart) {
        startCharDepth++;
    }
    if(temp == objEnd && valuesFound) {
        if(temp != objStart) {
            startCharDepth--;
        }
        //if we've reached the final level of end character, break out of loop
        if(startCharDepth <= 0  || (temp == objStart && startCharDepth % 2 == 0)) {
            count++;
```

Once the first instance of a valid opening char is identified - the respective closing char is saved and kept track of as we traverse through whatever data we're trying to get through.

Most of the uses for JSONHelper are for the individual class deserialization methods. Primarily using the "getValueFromKey" method, given a particular key name, JSONHelper will return the value associated with it in the serialized string. This means order really doesn't matter when accessing the data since you can get all the necessary fields in the same way one

Use Cases

Serialization is an incredibly useful library function to have - allowing the easy transmission of object data in char* format. This implementation could be used to save a dataframe to disk, turning it into a char* and writing to a text file. Then, to access the data we would just need to read in the file and re-interpret. This would allow our database to save state across uses, albeit in a simplistic way.

Serialization will also be useful in the transmission of data. If we can take the data structures or data frame pieces we want to transmit to another server or a backup and convert to a char*, it will be far easier to send and receive. The client will simply deserialize at the other end.

Pitfalls

While we do believe the readability of our serialized objects is the largest benefit of this serialization API, it was also the hardest to implement. As mentioned above, we already took some liberties with overall design ignoring fencepost commas and wrapping every primitive value in single-quotes. Also, abstracting the code necessary to build a JSON object piece-by-piece can make some of the Serializable write methods somewhat difficult to parse. Our client classes benefit from being able to use an interface that is as simplistic as some of the byte stream serialization implementations, but the code supporting that ease of use requires a lot of string-building behind the scenes. Particularly when dealing with parsing array values and finding a specific pair of quotes required an amount of debugging in the JSONHelper class.