Introduction

Serializing objects to and from json is tedious and error prone, potentially taking away from time doing real dev work. This library allows you to easily serialize and deserialize objects to and from json. This is especially useful for networking in games, but the addition of runtime editable constants allows the game designer to tweak values during runtime, especially handy for tweaking health/damage systems or normalizing object speeds.

How to use this code

Unsupported Member Types

- Pointers to objects
- Arrays of any kind [std::vector]
- C-style strings [const char*] (can be serialized, but not deserialized)
- Complex Data structures in the standard library eg. unordered_map
- Static variables
- Objects you do not have source code access to
- Objects without default constructors are not supported for runtime editability, as a default instance of the class is serialized to produce a default json Schema.

Make your Class Json Serializable

The following is an example layout of a json serializable class

```
#include "MetaSystem/MetaSystem.h"

//Object.h
Class Object
{
public:
    META_DATA(Object); //registers member functions for metasystem, public section
private:
     Int x;
     Vector2D position;
     Bool exampleBool;
}
-------------------------------------------------------------------------
//Object.cpp
DEFINE_META(Object) //this simply registers each member to the metasystem
{
    ADD_MEMBER(x);
    ADD_MEMBER(position);
    ADD_MEMBER(exampleBool);
}
```

And that's it! They are the only requirements to be able to serialize and deserialize your object from json strings. No manual serialization/deserialization!

Serialize your Object to Json

This is the easy part

```
#include "MetaSystem/MetaSystem.h"

Object x;
Std::cout << Variable(&x).ToJson(); //serialization, in 1 step
-------------------------------------------------------
//output json
{
  "x" = 7,
  "Position" = {
         "X" : 50,
         "Y" : 100
   },
   "exampleBool" : true
}
```

Deserialize your Object from Json

Given the previous example json, all you need to do is create a dummy variable you want to assign the json values to, and then call the following

```
Object x;
Variable(&x).FromJson<Object>(json);
//its as easy as that
```

Object x will then have all of the properties present in the json.

Hide Warning Messages

In release mode I have macros to ensure that debug messages corresponding to the MetaSystem are not printed to the console. However if you want to hide these messages in debug mode you can override them by giving the following macro definition.

```
#define META_DEBUGGING false
```

Make your Objects Runtime Editable

To make your properties runtime editable via json you will have to add a little more to your object structure.

```
#include "MetaSystem/MetaSystem.h"

//Object.h
Class Object : public RuntimeEditable<Object> //notice the use of CRTP
{
public:
    META_DATA(Object); //registers member functions for metasystem, public section
private:
     Int x;
     Vector2D position;
     Bool exampleBool;
}
-------------------------------------------------------------------------
```

```
//Object.cpp
DEFINE_META(Object) //this simply registers each member to the metasystem
{
    ADD_RE_MEMBER(x); //Special “Runtime Editable” Members i.e. RE
    ADD_RE_MEMBER(position);
    ADD_RE_MEMBER(exampleBool);
    LINK_TO_JSON(Object); //Another new line here
}
```

AutoLister

Since I needed the class anyway, I figure you ought to know how to use it if you need it. Essentially you just inherit from autolister like so

```
#include "MetaSystem/MetaSystem.h"

Class Object : public AutoLister<Object>
{
}
```

Now every instance is added to the autolister on construction, and removed on deletion. This pattern is a very easy way of decoupling game components. To get access to the list, I’ve made a handy helper function

```
EntityClass::get<Object>(); //returns list of all instances of type <Object>
```

Very useful!

Features

Reload variable data at runtime

I had to choose an implementation method that made sense, as associating dynamic objects with their own editable json is hard to grasp from a user perspective. Instead I have created a RE_Member which stands for “Runtime Editable” member, which is associated with every instance of a particular object. For example, object specific variables like a bullet’s speed, a character’s animation key, or a system boolean e.g. for toggling simulation of network conditions. These members when added to an objects metadata (as described above) will allow the production of a default json schema. Editing this json file will propagate changes to game objects at runtime, and also maintain its state through multiple re-runs of the executable. If the json schema ever gets messed up, or you need to change the schema. Simply make the necessary changes in C++ and delete the old schema file. A new one will be replaced on the next execution with up to date members and a valid schema.

Serialize Objects to Json with Ease

Serializing any generic primitive and nested serializable objects require a lot of hard work if it wants to be easy to use. My implementation requires only 1 line of code to actually serialize an object to json.

Deserialize Objects from Json with Ease

Deserializing any generic primitive and nested serializable objects require a lot of hard work if it wants to be easy to use. My implementation requires only 1 line of code to actually deserialize an object from json.

Automatic Instance Lister Class

As mentioned earlier in the user guide, AutoLister is a special class to keep track of object instances and allow decoupled access for objects to use them. It adds object to the list on construction, removes them on destruction, and provides an easy interface to access them anywhere in your code at a later time.

FileMonitor

FileMonitor is a really simple class that can allow you to associate a file with a callback function. When the file changes in any way, it invokes the callback function, and resets the new time.

Demonstration

- Game starts with a SpinningBox object.
- Game will produce a json file with all editable attributes for you to edit, including position and color objects.
- Editing the variables inside the json file will propagate to the game objects associated with that json.
- Any changes to the json file will be persisted when re-running the application.