C++ data serialization

with JSON examples

Volodymyr Bahrii 2020-04-13

Goals

- Create a type MetaProperty which can be easily serialized and deserialized (printed to console or a file, converted into a JSON object or a SQL request)
- Create utils functions to help distinguish MetaProperty and struct which contains MetaProperty at compile time
- Create helper classes for serialization and deserialization(if possible) of types which contain MetaProperty types (STL and custom containers, structures)
- Show examples

Third party libraries

- magic_get (Boost.PFR) - used for converting C++ struct to an std::stuple
  struct { int, std::string}; -> std::tuple<int, std::string>
  https://github.com/apolukhin/magic_get

- nlohmann (JSON library) - used in JSON serializer code
  https://github.com/nlohmann/json

Compile time string type

This type will be used for storing the name of each serializable value

```
template<char ... Chars>
struct CompileTimeString {
    static constexpr const char value[sizeof...(Chars)+1] = {Chars...,'\0'};
    static constexpr int size = sizeof...(Chars);
};

template<char ... Chars>
constexpr const char CompileTimeString<Chars...>::value[sizeof...(Chars)+1];

template<typename CharT, CharT ...String>
constexpr CompileTimeString<String...> operator"" _cts()
{
    return CompileTimeString<String...>();
}
```

Meta property

Name - a compile time string, T - a value type (int, string, bool…)

```
template <class T, class Name>
struct MetaProperty {
private:
    Name name;
public:
    T value;
    BHR_SERIALIZABLE

    MetaProperty() = default;
    MetaProperty(const T& inValue) : value(inValue) {}

    constexpr const char* GetName() const { return name.value; }
    operator T&() { return value; }

    T& operator=(const T& inValue) {
        value = inValue;
        return value;
    }
};
```

Meta property

specialization for a string value

```
template <class Name>
struct MetaProperty<std::string, Name> {
private:
    Name name;
public:
    std::string value;
    BHR_SERIALIZABLE

    MetaProperty() = default;
    MetaProperty(const char* inValue) : value(inValue) {}

    constexpr const char* GetName() const { return name.value; }
    operator std::string&() { return value; }
    operator const char* () { return value.c_str(); }

    std::string& operator=(const char* inValue) {
        value = inValue;
        return value;
    }
};
```

Macro helpers

```
// private
#define __BHR_STRUCT_NAME__ __bhr_serializable_struct__
#define __BHR_VAR_NAME__ __bhr_serializable__

// public
#define BHR_SERIALIZABLE_STRUCT char __BHR_STRUCT_NAME__;
#define BHR_SERIALIZABLE char __BHR_VAR_NAME__;

#define BHR_CTS(name) decltype(#name##_cts) // CTS - Compile Time String
#define BHR_TYPE(Type, Name) bhr::MetaProperty<Type, BHR_CTS(Name)> Name
#define BHR_TYPE_INITED(Type, Name, Value) BHR_TYPE(Type, Name) = Value
```

SFINAE helpers

```
struct sfinae_base {
  typedef char yes[1];
  typedef char no[2];
};

template <typename T>
class is_serializable_struct : public sfinae_base
{
    template<typename C> static yes& test( decltype(std::declval<C>().__BHR_STRUCT_NAME__) );
    template<typename C> static no& test(...) {}
public:
    static bool const value = sizeof(test<T>(0)) == sizeof(yes);
};

template <typename T>
class is_serializable : public sfinae_base
{
    template<typename C> static yes& test( decltype(std::declval<C>().__BHR_VAR_NAME__) );
    template<typename C> static no& test(...) {}
public:
    static bool const value = sizeof(test<T>(0)) == sizeof(yes);
};
```

Using MetaProperty

```cpp
#include "types/Serializable.h"

struct User
{
    MetaProperty<std::string, decltype("name"_cts)> name;
    MetaProperty<int, BHR_CTS(age)> age;
};


struct UserWithMacro
{
    BHR_TYPE(std::string, name);
    BHR_TYPE_INITED(int, age, 18);
};
```

```cpp
User user{"Volodymyr", 31};
std::cout << user.name.GetName()
          << ": " << user.name
          << ", " << user.age.GetName()
          << ": " << user.age
          << std::endl;


UserWithMacro user2{"Olesia", 28};
std::cout << user2.name.GetName()
          << ": " << user2.name
          << ", " << user2.age.GetName()
          << ": " << user2.age
          << std::endl;
```

Application Output:
name: Volodymyr, age: 31
name: Olesia, age: 28

JSON serializer public API

```
class NlohmannSerializer
{
  using OutSerialized = nlohmann::json;

public:
  template <class InSerializable>
  static void serialize(OutSerialized& outSerialized, const InSerializable& inSerializable)
  {
    auto tmpTuple = boost::pfr::structure_to_tuple(inSerializable);
    std::apply([&outSerialized](auto&&... elem)
    {
            ((serialize_impl(outSerialized, elem)), …); // process tuple elements
    }, tmpTuple);
  }

  template <class InSerializable>
  static OutSerialized serialize(const InSerializable& inSerializable)
  {
    OutSerialized outSerialized;
    serialize(outSerialized, inSerializable);
    return outSerialized;
  }
…
```

JSON serializer methods

- ***serialize_impl* -** skips all non-serializable types. Invokes a top-level *serialize()* method for nested serializable structures and a *serialize_key_value()* method for other types
- ***serialize_array_value*** - adds a received value into a JSON array (does recursive call of *serialize()* for nested data structures)
- ***serialize_key_value*** - a family of methods for serialization different types including STL containers (can be easily extended if needed).

JSON serializer implementation

```
template <class T>
static constexpr void serialize_impl(OutSerialized& outSerialized, const T& inSerializable)
{
    if constexpr (sfinae_utils::is_serializable<T>::value)
    {
        if constexpr(sfinae_utils::is_serializable_struct<decltype(std::declval<T>().value)>::value)
            serialize(outSerialized[inSerializable.GetName()], inSerializable.value); // recursion
        else
            serialize_key_value(outSerialized, inSerializable.GetName(), inSerializable.value);
    }
}
```

JSON serializer implementation

work horses

```
template <class T>
static void serialize_array_value(OutSerialized& outSerializedArray, const T& inValue)
{
    if constexpr (sfinae_utils::is_serializable_struct<T>::value)
    {
        nlohmann::json obj;
        serialize(obj, inValue); // recursion
        outSerializedArray.push_back(obj);
    }
    else
        outSerializedArray.push_back(inValue);
}

template <class T>
static constexpr void serialize_key_value(OutSerialized& outSerialized,
                                          const std::string& inName,
                                          const T& inValue)
{
    outSerialized[inName] = inValue;
}
```

JSON serializer implementation

adapters

```
template <class T, class U>
static constexpr void serialize_key_value(OutSerialized& outSerialized,
                                          const std::string& inName,
                                          const std::map<T, U>& inSerializable)
{
    for (const auto& elem : inSerializable)
    {
        serialize_key_value(outSerialized[inName], elem.first, elem.second);
    }
}

template <class T>
static constexpr void serialize_key_value(OutSerialized& outSerialized,
                                          const std::string& inName,
                                          const std::vector<T>& inSerializable)
{
    outSerialized[inName] = {};
    for (const auto& elem : inSerializable)
        serialize_array_value(outSerialized[inName], elem);
}
```

JSON array serializer implementation

adapters

```
template <class T>
static constexpr void serialize_key_value(OutSerialized& outSerialized,
                                          const std::string& inName,
                                          const std::list<T>& inSerializable)
{
    serialize_key_value(outSerialized, inName,
                        std::vector<T>(inSerializable.begin(), inSerializable.end()));
}

template <class T>
static constexpr void serialize_key_value(OutSerialized& outSerialized,
                                          const std::string& inName,
                                          const std::set<T>& inSerializable)
{
    serialize_key_value(outSerialized, inName,
                        std::vector<T>(inSerializable.begin(), inSerializable.end()));
}
```

Example #1 - primitives

```
struct User
{
    BHR_TYPE(int, id);
    BHR_TYPE(std::string, mail);
    BHR_TYPE(std::string, password);

    BHR_SERIALIZABLE_STRUCT
};
```

```
User user {1, "user@gmail.com", "qw!@3$"};
auto result = NlohmannSerializer::serialize(user);
std::cout << result << std::endl;
```

Application Output:

{"id":1,"mail":"user@gmail.com","password":"qw!@3$"}

Example #2 - nested data structures

```
struct Location
{
    BHR_TYPE(double, latitude);
    BHR_TYPE(double, longitude);
    BHR_TYPE(int, altitude);
    BHR_SERIALIZABLE_STRUCT
};

struct NamedLocation
{
    BHR_TYPE(std::string, name);
    BHR_TYPE(Location, location);
    BHR_SERIALIZABLE_STRUCT
};

struct Person
{
    BHR_TYPE(std::string, name);
    BHR_TYPE(int, age);
    BHR_SERIALIZABLE_STRUCT
};
```

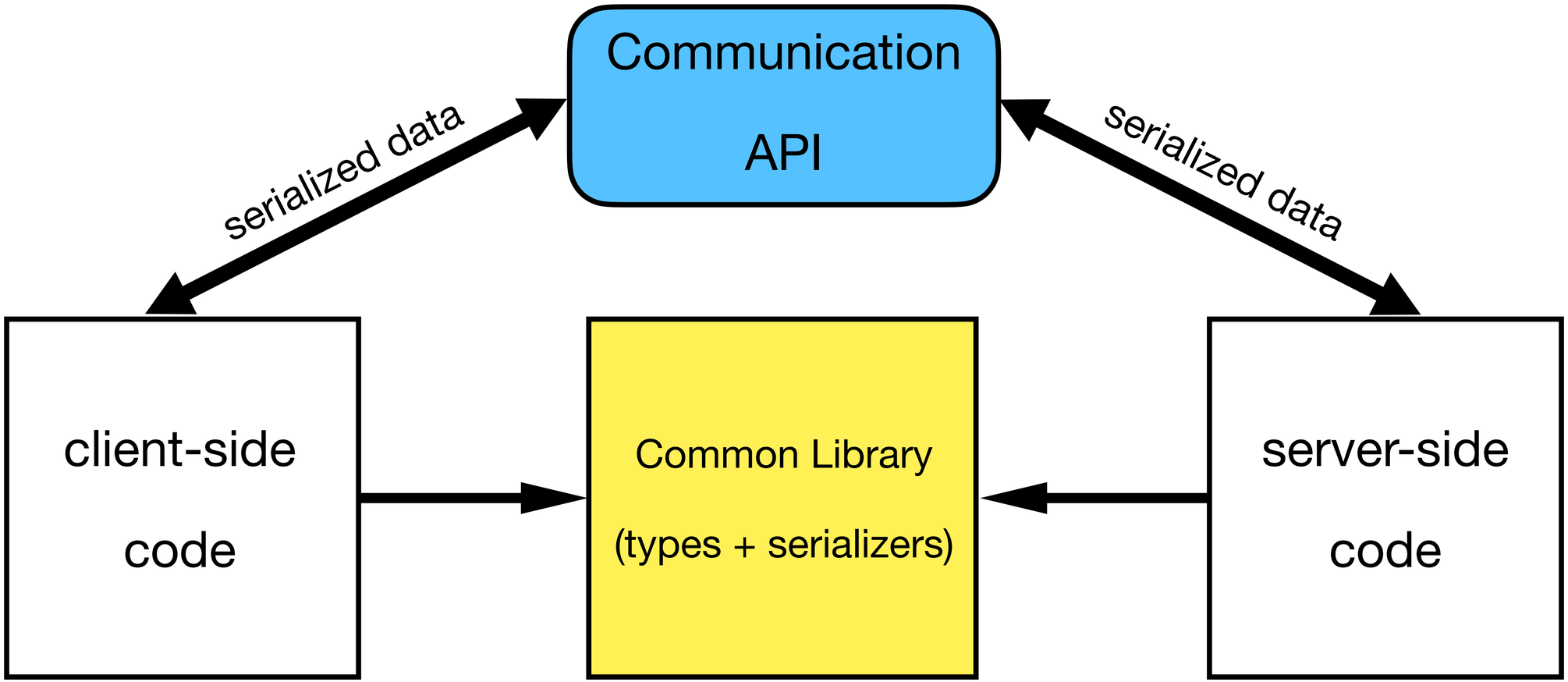
```
struct Trip {
    BHR_TYPE(std::string, title);
    BHR_TYPE(Person, person);
    BHR_TYPE(std::vector<NamedLocation>, route);
    BHR_SERIALIZABLE_STRUCT
};


//// USAGE /////
Trip trip = {
  "my trip”,
  Person{"Volodymyr", 31},
  std::vector<NamedLocation>{
    NamedLocation{"home", Location{50.437, 30.522, 143}},
    NamedLocation{"office", Location{50.501, 30.493, 100}}
  }
};

auto result = NlohmannSerializer::serialize(trip);
std::cout << result << std::endl;
```

Application output


```
{
   "person":{
      "age":31,
      "name":"Volodymyr"
   },
   "route":[
      {
         "location":{
            "altitude":143,
            "latitude":50.437,
            "longitude":30.522
         },
         "name":"home"
      },
      {
         "location":{
            "altitude":100,
            "latitude":50.501,
            "longitude":30.493
         },
         "name":"office"
      }
   ],
   "title":"my trip"
}
```

Client-server data exchange example

client and server are using common library which does data serialization/deserialization



Questions

- Q: why a MetaProperty name is a compile time string?
  A: the name must be a part of a type (cannot be changed in runtime)

- Q: Why there is no adapters for std::multiset, std::unordered_map and others?
  A: It is an example how MetaProperty can be user. You can easily support other types by adding **serialize_key_value** methods.

- Q: Where I can use it?
  A: Any place when need to serialize/deserialize custom data structures (nested structures as well).

- Q: Can I use mixed types in my custom struct (serializable and non-serializable types)
  A: Yes, non-serializable types will be skipped

More info

- github: https://github.com/skident/BhrSerializer
- e-mail: bagriy.volodymyr@gmail.com