JSonSerial2
C++ Object Serialization in JSON

- JsonSerial allows serializing **a set of C++ objects** (a single object, a collection, a **cyclic** or acyclic graph of objects)
- In most cases (and contrary to some other tools), JsonSerial **does not require writing read/write functions**. Instead, you just need to register which classes and members you want to serialize.
- JsonSerial can handle basic types, enums, C and C++ strings, plain C++ objects, most **C++ containers**, C arrays with brackets, **raw** and **smart pointers**.
- JsonSerial supports **graphs of objects pointing to each other**, including **cyclic graphs**. Pointees are automatically created when reading a JSON file. **Shared objects** (objects pointed by multiple pointers) are **not duplicated** (this feature is optional).
- JsonSerial supports **single and multiple inheritance** and **class polymorphism**. When needed, the class of derived classes is stored in the JSON file so that the proper objects will be created when reading the JSON file.
- The names in the JSON file can be **whatever you want** (rather than auto-generated arbitray names), e.g. the names of the objects' members.
- The JSON syntax can be **relaxed**: comments are supported, quotes and commas can optionally be omitted.
- JsonSerial requires **UTF8**. It relies on **C++ 11 templates** and only consists of **header files**.

Author: Eric Lecolinet – Télécom Paris – Institut Polytechnique de Paris
http://www.telecom-paris.fr/~elc (firstname.lastname@telecom.paris.fr)

Content

Basic features

This example shows how to serialize **basic types** and **enums**, C++ **strings**, C++ **lists** and **plain** objects. Below is the class that will be serialized.

```
class Contact {
public:
  enum Gender {Unknown, Male, Female, Other};
  struct Phone {std::string type, number;};

  friend JSonClasses* registerClasses();  // needed if variables are not public!

  Contact() = default;                          // needed to instantiate pointees!
  Contact(std::string const& firstname, std::string const& lastname,
          unsigned int age, Gender gender);

  void addPhone(std::string const& type, std::string const& number);
  void print(std::ostream& out) const;

private:
  std::string firstname, lastname;
  unsigned int age{};
  Gender gender;
  std::list<Phone> phones;  // list of plain objects
};
```

The **registerClasses**() function will be used to register the fields that need to be serialized. This function must be a **friend** of the class if its variables are not public. The **no-argument constructor** will be needed to instantiate pointees. We'll in another section how to relax these constraints.

Below is the source code for serializing the class. First, we must include the appropriate **headers:**

- **jsonserial.hpp** must be included **first**
- Then the header(s) for serializing C++ containers (here **std::list** is used, thus **jsonserial_list.hpp** must be included)
- **using namespace jsonserial** avoids prefixing JSonSerial classes their namespace (i.e. by **jsonserial::**)

```
#include "jsonserial.hpp"
#include "jsonserial_list.hpp"
using namespace jsonserial;
```

The **registerClasses**() function registers the classes and class members that need to be serialized:

```
JSonClasses * registerClasses() {
  auto * classes = new JSonClasses();

  classes->addClass<Contact>("Contact")
  .addMember("firstname", &Contact::firstname)
  .addMember("lastname", &Contact::lastname)
  .addMember("gender", &Contact::gender)
  .addMember("age", &Contact::age)
  .addMember("phones", &Contact::phones);
```

```
  classes->addClass<Contact::Phone>("Phone")
  .addMember("type", &Contact::Phone::type)
  .addMember("number", &Contact::Phone::number);

  return classes;
}
```

This function creates and returns an instance of **JSonClasses**, which **addClass**() method registers a class:

```
classes->addClass<Contact>("Contact")
```

The template argument of **addClass**() is the C++ class, its variable argument is a UTF8 string that should be the name of the class.

Members are registered by calling the **addMember**() method of the registered class:

```
.addMember("firstname", &Contact::firstname)
```

The first argument is a UFT8 string that identifies this member in the JSON file. Typicially it's the name of the variable, but another name can be used provided that two members of the same class don't have the same name. The second argument is a pointer to the corresponding class member (note the **& sign**).

As previouly said, **registerClasses**() must be a **friend** of the serialized class, otherwise it won't have access to class members and the program won't compile.

Let's now suppose that two objects, **alice** and **bob** have been created:

```
Contact* bob = new Contact("Bob", "Dûpontèle", 31, Contact::Male);
bob->addPhone("home", "123 456-7890");
bob->addPhone("mobile","703 221-2121");

Contact* alice = new Contact("Alice", "Dûpontèle", 33, Contact::Female);
alice->addPhone("home", "123 456-7890");
alice->addPhone("office", "211 1234-1234");
```

A **std::list** pointing to these 2 objects can be serialized and deserialized as follows:

```
std::list<Contact*> users{bob, alice};

static JSonSerial js(registerClasses());

if (!js.write(users, "data.json", false)) return;  // serializes users

std::list<Contact*> users2;
if (!js.read(users2, "data.json") return;  // deserializes into users2
```

js is a **JSonSerial** object that can (de)serialize any **instance** of the classes registered by registerClasses(), or a C++ **container** containing instances or pointing to them.

This object does not need to be **static**, but making it static allows reusing it if instances are (de)serialized multiple times.

The arguments of the **write**() methods are:

1. The object to be serialized, either a **plain object**, a **raw** or **smart pointer**, or a **container** (std::list in this example). Not only this object, but also **all the objects it contains or points to**, will be serialized (provided that the corresponding variables were registered)
2. The **file name** or a **std::ostream**
3. **A boolean value** specifying whether the objects' graph is **cyclic**. As it not the case in this example, its value can be false (this feature will be explained in more detail in the next example).

The arguments of the **read**() methods are:

1. The object to be deserialized, either a **plain object**, a **raw** or **smart pointer**, or a **container**. Not only this object, but also **all the objects it contains or points to** in the JSON file will be deserialized. There is no need to specify whether the graph is cyclic or not.
2. The **file name** or a **std::istream**

Both methods return false in case of an **error**. By default, errors are printed out on **std::cerr.**

Errors can also be displayed in a user-specific way by providing a **error handler** to the **JSonSerial** constructor (similarly, a error handler can be provided to the **JSonClasses** constructor). See the *Other features* section.

The **std::list** could also be serialized to / deserialized from a stream:

```
void test_stream() {
  // same as above...

 std::stringstream ss;

  if (!js.write(users, ss, false)) return;  // serializes users
  std::cout << ss.str() << std::endl;
  if (!js.read(users2, ss) return;  // deserializes into users2
}
```

In addition, the data writen into the stringstream (which is the **generated JSON**) is printed on the console, which is convenient for testing.

Full source code

Generated JSON File:

```
[
  {
    "firstname": "Bob",
    "lastname": "Dûpontèle",
    "gender": 1,
    "age": 31,
    "phones": [
      {
        "type": "home",
        "number": "123 456-7890"
      },
      {
        "type": "mobile",
        "number": "703 221-2121"
      }
    ]
  },
  {
    "firstname": "Alice",
    "lastname": "Dûpontèle",
    "gender": 2,
    "age": 33,
    "phones": [
      {
        "type": "home",
        "number": "123 456-7890"
      },
      {
        "type": "office",
        "number": "211 1234-1234"
      }
    ]
  }
]
```

Shared objects and cyclic graphs

This second example is similar to the previous one, except that **Contact** now has pointers and a **std::map** that point to other **Contact** objects:

- **partner** points to the Contact's partner,
- **parent1** and **parent2** point to his/her parents,
- **children** point to his/her children

This example involves **shared objects** and a **cyclic graph**: not only several pointers can point to the same object, but objects point to each other. JSonSerial allows solving both problems and can deserialize and create objects exactly as they were before writing them.

```
class Contact {
public:
  enum Gender {Unknown, Male, Female, Other};
  struct Phone {std::string type, number;};

  friend JSonClasses* registerClasses();  // needed if variables are not public!

  Contact() = default;   // needed to instantiate pointees!
  Contact(std::string const& firstname, std::string const& lastname,
          unsigned int age, Gender gender);

  void addPhone(std::string const& type, std::string const& number);

  void linkPartners(Contact* p) {
    partner = p;
    if (p) p->partner = this;
  }

  void linkChildWithParents(Contact* p1, Contact* p2) {
    parent1 = p1;
    parent2 = p2;
    if (p1) p1->children[firstname] = this;
    if (p2) p2->children[firstname] = this;
  }

  void print(std::ostream& out) const;

private:
  std::string firstname, lastname;
  unsigned int age{};
  Gender gender;
  std::list<Phone*> phones;  // list of object pointers
  std::map<std::string, Contact*> children;   // map of object pointers
  Contact *partner{}, *parent1{}, *parent2{}; // pointers MUST be initialized!
};
```

Importantly, **pointers must be properly initialized** (i.e. they should be null or point to a valid object); the **read**() function may crash otherwise!

The **registerClasses**() function is similar except that we now have more members to register:

```
#include "jsonserial.hpp"
#include "jsonserial_list.hpp"
#include "jsonserial_map.hpp"
using namespace jsonserial;
```

```
JSonClasses* registerClasses() {
  auto* classes = new JSonClasses();

  classes->addClass<Contact>("Contact")
  .addMember("firstname", &Contact::firstname)
  .addMember("lastname", &Contact::lastname)
  .addMember("gender", &Contact::gender)
  .addMember("age", &Contact::age)
  .addMember("phones", &Contact::phones)
  .addMember("partner", &Contact::partner)
  .addMember("parent1", &Contact::parent1)
  .addMember("parent2", &Contact::parent2)
  .addMember("children", &Contact::children);

  classes->addClass<Contact::Phone>("Phone")
  .addMember("type", &Contact::Phone::type)
  .addMember("number", &Contact::Phone::number);

  return classes;
}
```

Let's now create some parents and children and link them together:

```
  Contact* bob = new Contact("Bob", "Dûpontèle", 31, Contact::Male);
  bob->addPhone("home", "123 456-7890");

  Contact* alice = new Contact("Alice", "Dûpontèle", 33, Contact::Female);
  alice->addPhone("home", "123 456-7890");
  alice->addPhone("office", "211 1234-1234");

  Contact* karim = new Contact("Karim", "Dûpontèle", 9, Contact::Male);
  karim->addPhone("mobile","122 122-1222");

  Contact* susan = new Contact("Susan", "Dûpontèle", 11, Contact::Female);
  susan->addPhone("mobile","133 133-1333");

  bob->linkPartners(alice);
  karim->linkChildWithParents(bob, alice);
  susan->linkChildWithParents(bob, alice);
```

A list (or any other standard container) containing or pointing to these 4 objects can be (de)serialized almost in the same way as in the previous example:

```
  std::list<Contact*> users{bob, alice, karim, susan};

  static JSonSerial js(registerClasses());

  if (!js.write(users, "data.json", true)) return;  // last arg must be true!

  std::list<Contact*> users2;
  if (!js.read(users2, "data.json") return;
}
```

Importantly, **the last argument of the write() function must be true** because the graph is **cyclic**.

This option both avoids **duplicating** shared objects and **infinite loops** in the presence of a cyclic graph. A **@id field** is then added to each user-defined object in the JSON file (see example of

generated data below). The fields of the objects are written only the first time they are encountered, then objects are refererred by their **id** in the JSON file.

Note that this feature only works with **user-defined** classes (classes registered by calling **addClass**()), but not with C++ strings or standard C++ containers.

[Full source code]

Generated JSON File:


```
[
  {
   "@id": "1",
   "firstname": "Bob",
   "lastname": "Dûpontèle",
   "gender": 1,
   "age": 31,
   "phones": [
    {
     "@id": "2",
     "type": "home",
     "number": "123 456-7890"
    },
    {
     "@id": "3",
     "type": "mobile",
     "number": "703 221-2121"
    }
   ],
   "partner": {
    "@id": "4",
    "firstname": "Alice",
    "lastname": "Dûpontèle",
    "gender": 2,
    "age": 33,
    "phones": [
     {
      "@id": "5",
      "type": "home",
      "number": "123 456-7890"
     },
     {
      "@id": "6",
      "type": "office",
      "number": "211 1234-1234"
     }
    ],
    "partner": "@1",
    "parent1": null,
    "parent2": null,
    "children": {
     "@id": "7",
     "Karim": {
      "@id": "8",
      "firstname": "Karim",
      "lastname": "Dûpontèle",
      "gender": 1,
      "age": 9,
      "phones": [
       {
        "@id": "9",
        "type": "mobile",
        "number": "122 122-1222"
       }
      ],
      "partner": null,
      "parent1": "@1",
      "parent2": "@4",
      "children": {
       "@id": "10",

      }
     },
     "Susan": {
      "@id": "11",
      "firstname": "Susan",
      "lastname": "Dûpontèle",
      "gender": 2,
      "age": 11,
      "phones": [
       {
        "@id": "12",
        "type": "mobile",
        "number": "133 133-1333"
       }
      ],
      "partner": null,
      "parent1": "@1",
      "parent2": "@4",
      "children": {
       "@id": "13",

      }
     }
    }
   },
   "parent1": null,
   "parent2": null,
   "children": {
    "@id": "14",
    "Karim": "@8",
    "Susan": "@11"
   }
  },
  "@4",
  "@8",
  "@11"
]
```

Inheritance and class polymorphism

This third example is similar to the previous one, except that **Contact** has a **PhotoContact** subclass. Moreover **PhotoClass** also derives from **Photo** (multiple inheritance), which is an **abstract** classs.

Contact is unchanged except that its print() method is **virtual** (and redefined in **PhotoContact**).

```
class Contact {
public:
  enum Gender {Unknown, Male, Female, Other};
  struct Phone {std::string type, number;};

  friend JSonClasses* registerClasses();

  Contact() = default;
  Contact(std::string const& firstname, std::string const& lastname,
          unsigned int age, Gender gender);

  void addPhone(std::string const& type, std::string const& number);
  void linkPartners(Contact* p);
  void linkChildWithParents(Contact* p1, Contact* p2);
  virtual void print(std::ostream& out) const;   // this method is now virtual.

private:
  // as in second example...
};


class Photo {  // Photo is an abstract class
public:
  friend JSonClasses* registerClasses();

  void setPhoto(const std::string& file);
  virtual void print(std::ostream& out) const = 0;  // abstract method

protected:
  std::string file;
};


class PhotoContact : public Contact, public Photo {  // Multiple inheritance
public:
  friend JSonClasses* registerClasses();

  PhotoContact() = default;
  PhotoContact(const std::string& firstname, const std::string& lastname,
               unsigned int age, Gender gender);

  void print(std::ostream& out) const override;
};
```

The **registerClasses**() function is similar except that we need to register **Photo** and **PhotoContact**.

Note that:

- Because **Photo** is an **abstract** class, the **addClass**() function must be called with a second argument that is **nullptr**. This means that this class cannot be instantiated.
- **PhotoContact** must specify that it **derives** from superclasses by calling **extends**(). Its template argument is the superclass. As many superclasses as needed can be specified in this way.
- Superclasses must be registered **before** subclasses (otherwise a "undeclared superclass" error will occur at runtime).

```
JSonClasses* registerClasses() {
  JSonClasses* classes = new JSonClasses();

  classes->addClass<Contact>("Contact")
  // as in previous example ...

  classes->addClass<Contact::Phone>("Phone")
  // as in previous example ...

 classes->addClass<Photo>("Photo", nullptr)    // abstract class => nullptr
  .addMember("file", &Photo::file);

 classes->addClass<PhotoContact>("PhotoContact")
  .extends<Contact>()                          // inherits from Contact
  .extends<Photo>();                           // inherits from Photo

  return classes;
}
```

A list (or any other container) pointing to both **Contact** and **PhotoContact** instances can then be (de)serialized as in the previous (i.e. second) example.

The name of the class will be saved in the JSON file using a special **@class** field, which will allow creating the same object when deserialing the file. Note however that this will work **only if the class is polymorphic**, i.e. if it has at least one virtual method.

Diamond inheritance and shadowed variables

Diamond inheritance works as expected when using **virtual class inheritance** or if the shared classes **don't contain variables**. It will be problematic otherwise as the class will contain several (inherited) variables with the same name, i.e. **shadowed variables**.

The same problem occurs if a class declares variables that have the same name as in its superclass. A simple solution just consists in giving them **different UTF8 names** when registering them by calling the **addMember**() function.

Full source code

Generated JSON File:


```
[
 {
  "@class": "PhotoContact",
  "@id": "1",
  "firstname": "Bob",
  "lastname": "Dûpontèle",
  "gender": 1,
  "age": 31,
  "phones": [
   {
    "@id": "2",
    "type": "home",
    "number": "123 456-7890"
   },
   {
    "@id": "3",
    "type": "mobile",
    "number": "703 221-2121"
   }
  ],
  "partner": {
   "@id": "4",
   "firstname": "Alice",
   "lastname": "Dûpontèle",
   "gender": 2,
   "age": 33,
   "phones": [
    {
     "@id": "5",
     "type": "home",
     "number": "123 456-7890"
    },
    {
     "@id": "6",
     "type": "office",
     "number": "211 1234-1234"
    }
   ],
   "partner": "@1",
   "parent1": null,
   "parent2": null,
   "children": {
    "@id": "7",
    "Karim": {
     "@class": "PhotoContact",
     "@id": "8",
     "firstname": "Karim",
     "lastname": "Dûpontèle",
     "gender": 1,
     "age": 9,
     "phones": [
      {
       "@id": "9",
       "type": "mobile",
       "number": "122 122-1222"
      }
     ],
     "partner": null,
     "parent1": "@1",
     "parent2": "@4",
     "children": {
      "@id": "10",

     },
     "photo": "karim.png",
     "width": 75,
     "height": 50
    },
    "Susan": {
     "@id": "11",
     "firstname": "Susan",
     "lastname": "Dûpontèle",
     "gender": 2,
     "age": 11,
     "phones": [
      {
       "@id": "12",
       "type": "mobile",
       "number": "133 133-1333"
      }
     ],
     "partner": null,
     "parent1": "@1",
     "parent2": "@4",
     "children": {
      "@id": "13",

     }
    }
   }
  },
  "parent1": null,
  "parent2": null,
  "children": {
   "@id": "14",
   "Karim": "@8",
   "Susan": "@11"
  },
  "photo": "bob.png",
  "width": 75,
  "height": 50
 },
 "@4",
 "@8",
 "@11"
]
```

Smart pointers

This example is similar to the second one except that **std::shared_ptr** and **std::weak_ptr** are used instead of raw pointers.

There is no much to say except that the linkPartners() and linkChildWithParents() methods must be written slightly differently as *this* if not a smart pointer (using *shared_from_this*() is another option).

```
using ContactPtr = std::shared_ptr<class Contact>;

class Contact {
public:
  enum Gender {Unknown, Male, Female, Other};
  struct Phone {std::string type, number;};

 friend JSonClasses* registerClasses();

  Contact() = default;
  Contact(std::string const& firstname, std::string const& lastname,
          unsigned int age, Gender gender);
  void addPhone(std::string const& type, std::string const& number);

  static void linkPartners(ContactPtr p1, ContactPtr p2)  {
    if (p1) p1->partner = p2;
    if (p2) p2->partner = p1;
  }

  static void linkChildWithParents(ContactPtr child, ContactPtr p1, ContactPtr p2) {
    if (child) {
      child->parent1 = p1;
      child->parent2 = p2;
      if (p1) p1->children[child->firstname] = child;
      if (p2) p2->children[child->firstname] = child;
    }
  }

  void print(std::ostream& out) const;

private:
  std::string firstname, lastname;
  Gender gender;
  unsigned int age{};
  std::list<Phone*> phones;
  std::weak_ptr<Contact> partner;
  std::shared_ptr<Contact> parent1, parent2;
  std::map<std::string, std::weak_ptr<Contact>> children;
};
```

The **registerClass**() function is as in the second example, so as the source code for (de)serializing the objects.

Example of source code for creating the objects:

```
ContactPtr bob =
std::make_shared<Contact>("Bob", "Dûpontèle", 31, Contact::Male);
bob->addPhone("home", "123 456-7890");
bob->addPhone("mobile","703 221-2121");

ContactPtr alice =
std::make_shared<Contact>("Alice", "Dûpontèle", 33, Contact::Female);
alice->addPhone("home", "123 456-7890");
alice->addPhone("office", "211 1234-1234");

ContactPtr karim =
std::make_shared<Contact>("Karim", "Dûpontèle", 9, Contact::Male);
karim->addPhone("mobile","122 122-1222");

ContactPtr susan =
std::make_shared<Contact>("Susan", "Dûpontèle", 11, Contact::Female);
susan->addPhone("mobile","133 133-1333");

Contact::linkPartners(bob, alice);
Contact::linkChildWithParents(karim, bob, alice);
Contact::linkChildWithParents(susan, bob, alice);

std::list<ContactPtr> users {bob, alice, karim, susan};
```

Full source code

Custom class/member registration

This example shows how to proceed when:
- The class **does not have a no-argument constructor**
- Adding a **registerClasses() function as a friend** is not possible
- The members must by read and written using their **setters/getters** or **custom user-specific functions.**

Obviously, the simplest solution is to add the missing methods to the class, but this is not always possible (e.g. because the class belongs to an existing library and can't be changed).

```
class Contact {
public:
  enum Gender {Unknown, Male, Female, Other};
  struct Phone {std::string type, number;};

  // NOTES:
  // - the class does not have a no-argument constructor
  // - registerClasses() is not declared as a friend, accessors will be used.
  // - there is no setChildren() method

  Contact(std::string const& firstname, std::string const& lastname,
          unsigned int age, Gender gender);

  void setFirstName(std::string const& name) {firstname = name;}
  std::string const& getFirstName() const {return firstname;}

  void setLastName(std::string const& name) {lastname = name;}
  std::string const& getLastName() const {return lastname;}

  void setGender(Gender value) {gender = value;}
  Gender getGender() const {return gender;}

  void setAge(unsigned int value) {age = value;}
  unsigned int const& getAge() const {return age;}

  void setPartner(Contact* p) {partner = p;}
  Contact* getPartner() const {return partner;}

  void setParent1(Contact* p) {parent1 = p;}
  Contact* getParent1() const {return parent1;}

  void setParent2(Contact* p) {parent2 = p;}
  Contact* getParent2() const {return parent2;}

  void addPhone(std::string const& type, std::string const& number) {
    phones.push_back(Phone{type, number});
  }

  void setPhones(std::list<Phone> const& phones) {this->phones = phones;}
  std::list<Phone> const& getPhones() const {return phones;}

  void addChild(Contact* child) {
    if (child) children[child->firstname] = child;
  }
```

```
  std::map<std::string,Contact*> const& getChildren() const {return children;}

  void print(std::ostream& out) const;

private:                                                  // as in second example
  std::string firstname, lastname;
  Gender gender;
  unsigned int age{};
  std::list<Phone> phones;
  std::map<std::string, Contact*> children;
  Contact *partner{}, *parent1{}, *parent2{};
};
```

In this version of the **registerClasses**() function:

- A **custom creator function** is provided as a second argument to **addClass**() because the class does not have a **non-argument constructor**. This function will instantiate the object when a pointee needs to be created by JSonSerial.
- Note that this argument can be **nullptr** if there is no need to create pointees (which is not the case in this example)
- Pointers to the **setter** and **getter** methods are provided to **addMember**() instead of pointers to variable members. Thus, the variables' values will be set by calling these methods. Note that this technique is less efficient (and can lead to problems in some tricky cases) because reading an object from the JSON file then involves making a **copy** from a temporary variable.
- This example also shows how to write **custom functions for reading and writing members**. This is needed here because the class only has an addChild() method but no setChildren() method. Such functions typically call the JSonSerial **readMember**() and **readMember**() methods.

```
JSonClasses* registerClasses() {
  JSonClasses* classes = new JSonClasses();

  classes->addClass<Contact>("Contact",
                             // a function is provided to create the instance
                             [](){return new Contact("", "", 0, Contact::Unknown);}
                             )
   // pointers to setters/getters are provided instead of pointers to members
  .addMember(("firstname", &Contact::setFirstName, &Contact::getFirstName)
  .addMember("lastname", &Contact::setLastName, &Contact::getLastName)
  .addMember("gender", &Contact::setGender, &Contact::getGender)
  .addMember("age", &Contact::setAge,  &Contact::getAge)
  .addMember("phones", &Contact::setPhones, &Contact::getPhones)
  .addMember("partner", &Contact::setPartner, &Contact::getPartner)
  .addMember("parent1", &Contact::setParent1, &Contact::getParent1)
  .addMember("parent2", &Contact::setParent2, &Contact::getParent2)

  // no setChildren() method => custom read/write functions are needed
  .addMember("children",
                                                  // reads children from JSON
            [](Contact& c, JSonSerial& js, std::string const& value) {
                std::map<std::string, Contact*> children;
                js.readMember(children, value);
                for (auto& it : children) c.addChild(it.second);
              },
```

```
                                                    // writes children to JSON
            [](const Contact& c, JSonSerial& js) {
                js.writeMember(c.getChildren());
               }
            );

  classes->add<Contact::Phone>("Phone")
  .addMember("type", &Contact::Phone::type)
  .addMember("number", &Contact::Phone::number);

  return classes;
}
```

While not illustrated here, it is also possible to provide a custom creator function to **addMember**(), either to instantiate an object or an element of a container. This feature allows using different creator functions depending on the member.

The previous example relies on **lambdas**, non-member or static functions could also be used.

The source code for (de)serializing the objects is the same as in the second example. Here is an example of source code for creating the objects:

```
  Contact* bob = new Contact("Bob", "Dûpontèle", 31, Contact::Male);
  bob->addPhone("home", "123 456-7890");
  bob->addPhone("mobile","703 221-2121");

  Contact* alice = new Contact("Alice", "Dûpontèle", 33, Contact::Female);
  alice->addPhone("home", "123 456-7890");
  alice->addPhone("office", "211 1234-1234");

  Contact* karim = new Contact("Karim", "Dûpontèle", 9, Contact::Male);
  karim->addPhone("mobile","122 122-1222");

  Contact* susan = new Contact("Susan", "Dûpontèle", 11, Contact::Female);
  susan->addPhone("mobile","133 133-1333");

  bob->setPartner(alice);
  bob->addChild(karim);
  bob->addChild(susan);

  alice->setPartner(bob);
  alice->addChild(karim);
  alice->addChild(susan);

  karim->setParent1(bob);
  karim->setParent2(alice);

  susan->setParent1(bob);
  susan->setParent2(alice);
```

Full source code

Other features

Post processing

Sometimes, some operations need to be performed **after reading or writing** the members of an object. In the next example, wasRead() and wasWritten() will be called after reading (resp. writing) a **Contact** object.

```
class Contact {
public:
  void wasRead();
  void wasWritten() const;
  // ...
};
```

In the registerClasses() function, one should then write:

```
   classes->addClass<Contact>("Contact")
     .postRead(&Contact::wasRead)          // called after reading an object
     .postWrite(&Contact::wasWritten);     // called after writing an object
```

Global and static variables

Global and **static variables** can also be serialized. In the ext example, globalStatus will appear in all **Contact** instances in the JSON file. Note that there is **no & sign** before the variable in this case:

```
static int global_status = 1;                          // static or global variable
```

In the registerClasses() function:

```
    classes->addClass<Contact>("Contact")
     .addMember("globalStatus", global_status);          // no & before variable
```

Primitive objects

Let suppose that **Float** is a class that embeds a float, **Text** a class that embeds a **std::string**, and that these classes allow setting and retreiving their value using the **= operator.** While these classes are not primitive types, they behave in a similar way. JsonSerial allows dealing with them as if they were **primitive types** by writing:

```
namespace jsonserial {
template <> struct is_integral<Float> : std::true_type {};
template <> struct is_string<Text> : std::true_type {};
}
```

is_integral<> (resp. **is_string<>**) mean that the objects of this class will be treated as an integral type (resp. a string) when (de)serialized by JSonSerial.

The same technique can be used for making JSonSerial to treat **subclasses of standard containers** as containers (by default they are considered as user-defined objects and must thus be registered through **addClass**()):

```
class Books : public std::list<std::string> {};

namespace jsonserial {
template <> struct is_std_list<Books> : std::true_type {};
}
```

Books will then be serialized in the same ways as a **std::list** container.

When subclassing **std::map** or **std::unordered_map** the [] operator must also be defined:

```
class Library : public std::map<std::string, Books*> {
public:
   std::string& operator[](std::string& key) {   whatever... }
};

namespace jsonserial {
template <> struct is_std_map<Library> : std::true_type {};
}
```

JSON syntax

By default, JsonSerial conforms to **JSON syntax** with some small differences:

- **/* ... */** and **// comments** are supported
- Name/value lists and arrays can have **trailing commas**
- Values can be **triple quoted** (ex: """let \t it \n be""") in which case they can contain double quotes, newlines and other control characters
- Names and values **cannot start with @** as this symbol is used for specifying classes and IDs.

The JSON syntax can be **relaxed** (or not) by calling **JsonSerial**::**setSyntax**() with, as an argument:

- JsonSerial::Strict: **strict syntax**: no option is allowed (comments are disabled)
- JsonSerial::Relaxed: **relaxed syntax**: all options are allowed

or an ORred combination of:

- JsonSerial::Comments: allows comments (the default),
- JsonSerial::NoQuotes: names and values can be unquoted (when non ambiguous)
- JsonSerial::NoCommas: name/value pairs can be separated by a comma or by a newline
- JsonSerial::Newlines: values can contain newlines and other control characters.

Error handling

By default, if an error is encountered when calling **addClass**(), **read**() or **write**() an error message is printed on JsonSerial. Error messages can be processed differently by providing a printing function to the **JSonSerial** and **JSonClasses** constructors:

```
JSonSerial js(registerClasses(),
              [](JSonError const& error){  custom processing of error }
              );
```

The variable members of the **JSonError** parameter are:

- the *type* of the error (an enum),
- *arg* (a string), an optional argument that is typically the name of the member,
- *fname* (a string), which is the filename, when available
- the *line* (an int) indicated the line where the error in the file (0 if N/A)

Limitations

JsonSerial:

- Requires a compiler that is **C++11 compliant.**
- Requires **UTF8**
- Does not support the JSON **\u notation** for specifying characters
- Names and values **cannot start with @** as this symbol is used for specifying classes and IDs.
- Provides reasonable performance but was not developed for storing huge object collections.
- May be slow to compile as it relies on non trivial template processing.

JsonSerial been developed and tested on MacOSX using clang and compiled/executed on https://rextester.com/ with gnu, clang and vc++ and on https://godbolt.org/ with gnu, clang and icc.