JsonSerial: C++ Object Serialization in JSON

JsonSerial provides a simple mean to **serialize C++ objects** (a single object, a collection or a graph of related objects). It does not require writing source code to read/write objects, this is done **automatically** through the magic of C++ templates. In most cases, it only requires declaring the classes and variables that need to be serialized.

JsonSerial supports basic types, enumerations, **raw** and **smart pointers**, user-defined **objects** (either **plain** or **pointed** objects), C and C++ **strings**, most C++ **containers** and C-style 1-dimensional **arrays** with brackets. When pointers point to objects the pointees are **automatically instanciated**.

Optionally, **JsonSerial** can detect **shared objects** (objects pointed by multiple pointers), which are then referenced by a **unique** ID in the JSON file instead of being duplicated. This feature allows storing **cyclic graphs** in JSON files.

JsonSerial also supports **single** and **multiple inheritance** and object **polymorphism**. When a pointer points to an object of a derived class, its **class name is stored** in the JSON file so that the proper object can be created when reading the file again.

JsonSerial consists of **header files** and relies on **C++ 11 templates**. **JsonSerial** has been test on MaxOSX and Linux (no guarantee it works on Windows).

JsonSerial Licence: GNU Lesser General Public License (LGPL) Version 3 or later.

First Example (without containers):

```
#include <jsonserial/jsonserial.hpp>                    // main JsonSerial header
#include <jsonserial/list.hpp>                        // for serializing std::list
using namespace jsonserial;                              // JsonSerial namespace

class Contact {
  friend void foo();                      // friend needed if variables not public
  std::string firstName, lastName;
  unsigned int age{0};
  Contact * partner{nullptr};
  std::list<Contact *> children;
public:
  // etc.
};

void foo() {
  JsonClasses cl;                                        // stores class declarations
  cl.defclass<Contact>("Contact")                 // declares Contact and its members
    .member("firstName", &Contact::firstName)
    .member("lastName", &Contact::lastName)
    .member("age", &Contact::age)
    .member("partner", &Contact::partner)
    .member("children", &Contact::children);

  JsonSerial js(cl);                                              // JSON reader/writer
  Contact contact;
  js.read(contact, "contact.json");            // reads contact and pointed objects
  js.write(contact, "contact-copy.json");     // writes contact and pointed objects
}
```

Including headers

The two first lines include the main **JsonSerial header** and the header(s) for serializing **standard containers** (here `std::list` because `children` is a list). There is one header for each C++ container (containers can't be serialized without including the corresponding JsonSerial headers). The third line avoids prefixing JsonSerial classes by their namespace (i.e. `jsonserial`).

The `foo()` function then declares the `Contact` class, and reads and writes a `Contact` object.

Declaring classes and members

`defclass()` declares a **C++ class**. As many classes as needed can be declared in this way. Its **template** argument is the *C++ class*, its **function** argument the *name* of the class (in UTF8).

`member()` declares a **member variable**. Its arguments are the **name of the variable** in the JSON file (in UTF8) and a **reference** to the variable. Any name can be used, but cannot start with @.

`member()` must have access to the object's variables, meaning that `foo()` must be a **friend** or a method of `Contact` (or that the variables are public). When not possible, `member()` can rely on **accessors** (see section *Declaring Accessors*). Finally, `member()` can also serialize **global** and **static** variables (see section *Global and static variables*).

Reading and writing a JSON file

The first argument of `read()` and `write()` is an object of a class that was declared using `defclass()`. This argument can be a **plain object** or a (raw or smart) **pointer**. The second argument is the *pathname* of the JSON file. The second argument can also be an input stream (for `read()`) or an output stream (for `write()`).

`read()` and `write()` return `true` on success. Otherwise they return `false` and an error is printed on `std::cerr`. Error handling can be **customized** and the JSON syntax can be **relaxed** (see sections *Error handling* and *JSON syntax*).

Pointed objects

The objects pointed by `partner` and `children` are **automatically created** when reading the JSON file (and so on if these objects themselves contain pointers). By default, pointed objects are created by calling the **no-argument** (or default) constructor of the class. When not possible or desirable, an ad hoc **creator function** can be given as an argument to `defclass()` or `member()` (see section *Constructors with arguments*).

Raw pointers must be properly **initialized** (i.e. be null or point to a valid object). Writing an object that has dangling pointer variables will crash the program!

Smart pointers. JsonSerial supports `std::shared_ptr` and `std::unique_ptr`. The latter can be used only if the object is pointed by only one pointer.

Null pointers. If a pointer is null in the JSON file, no object is created and the C++ pointer is set to `nullptr`.

Containers

JsonSerial supports:

1. One dimensional **C-style arrays** with brackets
2. `std::array`, `std::vector`, `std::list`, `std::deque`, `std::set`, `std::unordered_set`
3. `std::map` and `std::unordered_map` (but the key must be a `std::string` because JSON requires member names to be strings).

std **maps** are formatted as **JSON objects**. All other containers (and C-style arrays) are formatted as **JSON arrays**.

As said above, container headers must be **#included** after the main JsonSerial header (e.g. `"jsonserial/list.hpp"` after `"jsonserial/jsonserial.hpp"`). Otherwise, the program will **fail at runtime** with an error saying that the container class is unknown.

Second Example (with containers)

This example involves three classes: `Contact`, `PhoneNumber` and `Address`, which is a nested class of `Contact`. `Contact` has to members that are containers (`phoneNumbers` and `children`):

Header file `contact.hpp`

```
#include <string>
class PhoneNumber {
  friend class MyClasses;
  std::string type, number;
};

class Contact {
  friend class MyClasses;
  class Address {                                          // nested class
  public:
    std::string street, city, state, pcode;
  };
  std::string firstName, lastName;
  unsigned int age{0};
  enum {Unknown, Male, Female} gender{Unknown};
  bool isAlive{true};
  Address address;
  std::vector<PhoneNumber *> phoneNumbers;
  Contact *partner{nullptr}, *father{nullptr}, *mother{nullptr};
  std::list<Contact *> children;
public:
  // etc.
};
```

Implementation file `contact.cpp`

```
#include <iostream>
#include <jsonserial/jsonserial.hpp>
#include <jsonserial/list.hpp>
#include <jsonserial/vector.hpp>
#include "contact.hpp"
using namespace jsonserial;

class MyClasses : public JsonClasses {
public:
  MyClasses() {
    defclass<Contact>("Contact")
     .member("firstName", &Contact::firstName)
     .member("lastName", &Contact::lastName)
     .member("age", &Contact::age)
     .member("gender", &Contact::gender)
     .member("isAlive", &Contact::isAlive)
     .member("partner", &Contact::partner)
     .member("father", &Contact::father)
     .member("mother", &Contact::mother)
     .member("children", &Contact::children)
     .member("phoneNumbers", &Contact::phoneNumbers)
     .member("address", &Contact::address);
    // to be continued...
```

```
    defclass<PhoneNumber>("PhoneNumber")
     .member("type", &PhoneNumber::type)
     .member("number", &PhoneNumber::number);

    defclass<Contact::Address>("Contact::Address")            // nested class
     .member("street", &Contact::Address::street)
     .member("city", &Contact::Address::city)
     .member("state", &Contact::Address::state)
     .member("pcode", &Contact::Address::pcode);
  }

  static MyClasses& instance() {                              // static method
    static MyClasses cl;                                    // created only once
    return cl;
  }
};

void foo() {
  JsonSerial js(MyClasses::instance());          // JsonSerial reader/writer
  Contact contact;

  if (!js.read(contact, "contact.json")) return;         // reads the objects

  if (!js.write(contact, "contact-copy.json")) return; // writes the objects

  if (!js.write(contact, cout)) return;          // writes on standard output

  std::ostringstream ss;
  if (!js.write(contact, ss)) return;            // writes in a stream buffer
  std::cout << ss.str() << std::endl;
}
```

`MyClasses` is a subclass of `JsonClasses` that serves to declares all the C++ classes that are serialized. It has a static **`instance()`** method that declares the C++ classes only once, i.e., when it is called for the first time. As already explained, `MyClasses` must be a **friend** of the C++ classes (see an alternate solution below).

In this example, `address` is a **plain object**, it could also be a pointer. Conversely, the elements of the `children` and `phoneNumbers` containers are pointers but they could be **plain objects.** Finally, all pointers could be `std::shared_ptr` **smart pointers.**

Declaring accessors

Because `member()` must have access to the object's variables, `foo()` and `MyClasses` were **friends** of the serialized C++ classes in previous examples. While this is the simplest (and most efficient) solution, adding a **`friend`** statement is not always possible (e.g. for serializing existing classes which variables are private and that cannot be modified). JsonSerial can then still be used if the class has **public accessors**:

```
class PhoneNumber {                                  // not a friend of MyClasses
  std::string type, number;                          // private variables
public:
  const std::string& getType() const;                // public accessors
  const std::string& getNumber() const;
  void setType(const std::string&);
  void setNumber(const std::string&);
};
```

The `member`() method must then then be called with three arguments: the **name**, the **setter** and the **getter** of the variable. However, because this technique involves temporary variables it should be avoided when possible.

```
    defclass<PhoneNumber>("PhoneNumber")
     .member("type"&, &PhoneNumber::setType, &PhoneNumber::getType)
     .member("number"&, &PhoneNumber::setNumber, &PhoneNumber::getNumber);
```

Global and static variables

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

```
int globalStatus = 1;                                          // global variable

class MyClasses : public JsonClasses {
  static int objCount = 0;                                  // static class variable

  MyClasses() {
    defclass<Contact>("Contact")
     .member("globalStatus", global_var)                      // no & before variable
     .member("objCount", Contact::static_var)
     ...
  }
}
```

Single and Multiple inheritance

JsonSerial supports **single and multiple inheritance** . Suppose that `PhotoContact` derives from `Contact` and `Photo`:

```
class Photo {
  friend class MyClasses;
  std::string imagepath;
  unsigned int width, height;
public:
  Photo() : width(0), height(0) {}
};

class PhotoContact : public Contact, public Photo {
  friend class MyClasses;
  std::string location;
public:
  PhotoContact() {}
};
```

`PhotoContact` must then be defined as follows:

```
    defclass<PhotoContact>("PhotoContact")
     .extends<Contact>()                              // first superclass
     .extends<Photo>()                                // second superclass
     .member("location", &PhotoContact::location);
```

The template argument of `extends()` is the superclass. As many superclasses as needed can be specified in this way.

Diamond inheritance works as expected if **virtual** class inheritance is used or if the shared class(es) do(es) **not contain variables**. In contrast, non-virtual diamond inheritance of variables won't work because several variables will have the same name (see *shadowed variables* below).

Remarks:
- Superclasses must be declared **before subclasses** (otherwise a `undeclared superclass` runtime error will occur).
- In case of **shadowed variables** (variables having the same name in a subclass and a superclass), only the variable of the subclass is serialized. To serialize both, use different JSON names when declaring them with `member()`.
- The same problem occurs for non-virtual diamond inheritance, there is no way for solving it.

Abstract classes

Because **abstract classes** cannot be instanciated, they must be defined as follows. For instance, if `Photo` was an abstract class:

```
    defclass<Photo>("Photo", nullptr)
      // ... etc.
```

The `nullptr` argument specifies that JsonSerial **cannot instanciate** this class. Without this argument, runtime error `can't create instance of an abstract class` would occur.

Polymorphism

JsonSerial allows **polymorphic objects**. In the following example, `partner` is a `Contact` pointer which points to a `PhotoContact` object. A special `"@class"` member is then added to the JSON file when serializing this object, so that the same object will be created when deserializing the file:

```
class Contact {
  friend class MyClasses;
  Contact * partner = new PhotoContact();
  // ... etc.
public:
  virtual ~Contact() = default;                  // makes the class polymorphic
  // ... etc.
};
```

```
  "partner": {
    "@class": "PhotoContact",
    "imagePath": "Bessie.jpg",
    "with": 50,
    "height": 50,
    "firstName": "Bessie",
    "lastName": "Smith",
    ...
  }
```

Remarks:
- When present, `"@class"` is the first member of the JSON object. It only appears if the classes are **polymorphic** and if the pointer and the pointee have different classes.
- C++ classes are polymorphic if the base class contains **at least one virtual method** (here the destructor `~Contact()`).

Shared objects and cyclic graphs

JsonSerial optionally supports **shared objects** (objects pointed by several pointers) and **cyclic graphs**. For instance, in the second example, the two parents will point to each other (through their `partner` member) and they can also point to the same children, and vice-versa. JSON does not addresses these cases, which may cause duplications (in case of shared objects) or infinite loops (in case of cyclic graphs).

JsonSerial solves these problems when the `JsonSerial` instance is in ***sharing mode***, which is done by calling its `setSharing`() method **before** writing and reading objects. A special `"@id"` member is then added to all objects in the JSON file, as illustrated below. This makes it possible to manage shared objects properly when reading the file.

```
{
  "@id": "1",
  "firstname": "John",
  "lastname": "Smith",
  "mother": null,
  "father": null,
  "partner": {
    "@id": "2",
    "firstname": "Bessie",
    "lastname": "Smith",
    "mother": null,
    "father": null,
    "partner": "@1",
    "children": [
      {
        "@id": "3",
        "firstname": "Franck",
        "lastname": "Smith",
        "mother": "@2",
        "father": "@1",
        "partner": null,
        "children": []
      },
      {
        "@id": "4",
        "firstname": "Laura",
        "lastname": "Smith",
        "mother": "@2",
        "father": "@1",
        "partner": null,
        "children": []
      }
    ]
  },
  "children": [
    "@3",
    "@4"
  ]
}
```

Remarks:

- Only objects which class was declared with `defclass()` will be shared (basic types, strings and containers won't be shared).

Constructors with arguments

When reading a JSON file, objects that correspond to a class member that is a pointer are created by calling the **no-argument contructor** of their class (or its default constructor if the class has no constructor). The previous way of using `defclass()` **requires** the existence such a constructor (the source code won't compile otherwise). When not possible or desirable, an ad hoc **creator function** can be given as an argument to `defclass()` or `member()`.

Let's for instance suppose that `PhoneNumber` has a constructor that requires two arguments but no default constructor:

```
class PhoneNumber {
public:
  PhoneNumber(const std::string& type, const std::string& number);
  ...
};

class Contact {
  vector<PhoneNumber *> phoneNumbers;
  ...
};
```

A **Class Creator** can be provided to `defclass()` as follows (here using a lambda):

```
class MyClasses : public JsonClasses {
public:
  MyClasses() {
    defclass<PhoneNumber>("PhoneNumber", []{return new PhoneNumber("","");})
    ....
  }
}
```

Or, using a *static* method:

```
class MyClasses : public JsonClasses {
  static PhoneNumber * createPhoneNumber() {                  // static method
     return new PhoneNumber("","");
  }

public:
  MyClasses() {
    defclass<PhoneNumber>("PhoneNumber", createPhoneNumber)
    ...
  }
}
```

Class Creators can be *lambdas*, *static* class methods or *non-member* functions. They must have no parameter and return the newly created object (as a pointer).

An alternate solution is to provide a **Member Creator** to the `member()` function. This solution is more flexible as it allows creating pointees **differently** depending on each member:

```
class MyClasses : public JsonClasses {
public:
  MyClasses() {
    defclass<Contact>("Contact")
      .member("phoneNumbers", &Contact::phoneNumbers,
              [](Contact&) {return new PhoneNumber("","");})
      ...
  }
}
```

Member Creators can be 1) *instance* methods or 2) *lambdas*, *static* class methods or *non-member* functions that return the newly created object:

1. Instance methods have no parameter.
2. Other functions have one parameter that is a *reference* to the object containing the variable (e.g. `Contact&` in this example).

Remarks:

- If class and member creators are both specified, the more specific is used.
- *Custom read/write functions* can also be used for the same purpose (see below).

Custom read/write functions

Custom read/write functions allow reading/writing objets in a customized way, as in the following example:

```
class PhoneNumber {
  friend class MyClasses;
  std::string type, number;

  void readNumber(JsonSerial&, const std::string& s) {number = s;}
  void readType(JsonSerial&, const std::string& s) {type = s;}

  void writeNumber(JsonSerial& js) const {js.writeMember(number);}
  void writeType(JsonSerial& js) const {
     if (!type.empty()) js.writeMember(type);
  }
};
```

In this example, `writeType()` writes the `type` member only if it is not empty. `writeMember()` and `readMember()` are the standard `JsonSerial` methods for writing (resp. reading) a member.

The `member`() method must then be called with three arguments: the **name** of the member, then the functions fror **reading** and **writing** it:

```
   defclass<PhoneNumber>("PhoneNumber")
    .member("type", &PhoneNumber::readType, &PhoneNumber::writeType)
    .member("number", &PhoneNumber::readNumber, &PhoneNumber::writeNumber);
```

Read/Write functions can be 1) *instance* methods (as in this example) or 2) *lambdas*, *static* class methods or *non-member* functions:

- 1a) **Writing** instance methods (e.g. `writeType()`) have a `JsonSerial&` parameter and they are `const`
- 1b) **Reading** instance methods (e.g. `readType()`) have an additional `const string&`

parameter (which contains the value read by the parser) and are not `const`

- 2a) Other **writing** functions have a `JsonSerial&` parameter and a *const reference* to the object containing the variable
- 2b) Other **reading** functions have a `JsonSerial&` parameter, a *reference* to the object containing the variable and an additional `const string&` parameter.

Post processing

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

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

```
    defclass<Contact>("Contact")
     .postread(&Contact::wasRead)          // called after reading an object
     .postwrite(&Contact::wasWritten)      // called after writing an object
     // etc...
```

Post processing functions can be 1) *instance* methods (as in this example) or 2) *lambdas*, *static* class methods or *non-member* functions:

- 1a) **postread** instance methods have no parameter,
- 1b) **postwrite** instance methods have no parameter and they are *const*,
- 2a) Other **postread** functions have one parameter that is a *reference* to the object containing the variable (e.g. `Contact&`),
- 2b) Other **postwrite** functions have one parameter that is a *const reference* to the object containing the variable (e.g. `const Contact&`).

Container subclasses

By default, **container subclasses** are considered as user-defined objects (meaning they must be declared by using `defclass()` to allow them being serialized). This can be changed as follows:

```
class Books : public std::list<std::string> {};
template <> struct is_std_list<Books> : std::true_type {};
```

`Books` will then be considered as a `list` container, and serialized in the same way (provided that the `jsonserial/list.hpp` header way included)

In addition, objects deriving from `std::map` or `std::unordered_map` also require defining `operator[]`:

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

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

JSON syntax

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

- `JsonSerial::read()` **ignores comments** inside **/*** and ***/** and after **//**. This can be changed by calling `JsonSerial::setSyntax()` (see below).
- Name/value lists and arrays/containers 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 @** because this symbol has a special meaning (see previous sections on *Polymorphism* and *Shared objects*).

The JSON syntax can be **relaxed** by calling `JsonSerial::setSyntax()` with the following values:

- JsonSerial::Strict: **strict syntax**: no option is allowed (comments are thus 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: allows **newlines** and other control characters.

Error handling

By default, if an error is encountered when calling `defclass()`, `read()` or `write()` an error message is printed on `std::cerr`. Error messages can be processed differently if needed:

```
void errorHandler(const JsonError& e) {
   std::cerr << "Error in MyProgram: "; e.print(std::cerr);
}

MyClasses MyClasses::instance(errorHandler);

void foo() {
   JsonSerial js(MyClasses::instance, errorHandler);
   ...
}
```

Or, using a lambda:

```
void foo() {
  ostringstream ss;

  JsonSerial js(MyClasses::instance,
                [&ss](const JsonError& e){e.print(ss);});

  if (!js.read(contact, "contact.json")) {
    std::cerr << "Error in MyProgram: " << ss.str() << std::endl;
    return;
  }
}
```

The `errorHandler()` fonction can be a *lambda*, a *static* class method or a *non-member* function. It has a `JsonError` argument, which is an object that contains:

- the **type** of the error (an enum),
- **where** it occured (a string),
- **arg** (a string), an optional argument that is typically the name of the member,
- the **line** (an int) where it occured in the JSON file (if applicable, otherwise line = 0),
- the **what**() method, which returns the error string corresponding to **type**,
- the **print**() method, which prints this error on a `std::ostream` (this method is called by default to print on `std::cerr`).

See class `JsonError` in `jsonerror.hpp` for more details.

Limitations

JsonSerial:

- Requires a compiler that is **C++11 compliant.**
- Has been developed on MacOSX and tested on Debian with clang++ and g++. It should work on other recent Unix/Linux systems but has **not been tested on Windows**.
- Relies on **std::string**. Strings are usually encoded in **UTF8** on recent Unix/Linux systems, but not on Windows or older Unix systems, which can cause compatibility problems.
- Does not support the JSON **\u notation** in strings.
- Provides reasonable performance but was not developed for storing huge object collections. Making it easy to serialize C++ objects was the first concern.
- May be slow to compile as it relies on non trivial template processing.

Author/Contact

Eric Lecolinet – Télécom Paris
eric.lecolinet@telecom.paris.fr
http://www.telecom-paris.fr/~elc