Serialization in C++

How to save and load C++ objects?

What is serialization? What is it good for?

- Saving and loading C++ data structures:
  - Having a class: class A {int i1; };
  - Or an std::vector<A>
  - And a serializer object.
- Serialization is used for:
  - Persistence
  - Marshalling
  - Storing data in file
  - Sending messages
  - Reading configuration files

```
A a1;
Serializer.save(a1);
A a2;
Serializer.load(a2);

Std::vector<A> v1;
Serializer.save(v1);
Std::vector<A> v2;
Serializer.load(v2);
```

Serialization is trivial:

We have std::ostream and std::istream.

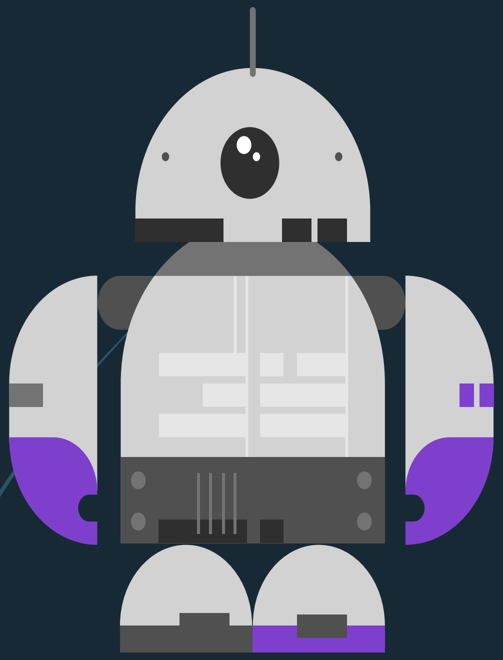
```
struct A {
    int i1; std::string s1;
};
std::ostream& operator<<(std::ostream& os, const A& a) {
    os << a.i1 << "," << a.s1; return os;
}
std::istream& operator>>(std::istream& is, A& a) {
    char ch; is >> a.i1 >> ch >> a.s1; return is;
}
A a1;
std::ofstream os("a.txt"); os << a1;
A a2;
std::ifstream is("a.txt"); is >> a2;
```

Serialization is more complicated

- The save() and load() functions are similar and must do the same.
- Different file formats (text, binary) require different operators.
- How to handle different versions of the class?
  - Add, rename, delete members
- How to handle:
  - polymorph objects?
  - pointers and smart pointers?
  - containers?
- Libraries, classes written by someone else?
  - You must write the stream operators for all types used in your code.
- Multiple languages?
- Stream operators are slow.

Serialization libraries

- Save / Load functions (code driven reading):
  - Boost serialization
  - Cereal (https://github.com/USCiLab/cereal)
- IDL (Interface Definition Language):
  Generates save / load functions from the IDL description.
  - ASN.1 (Abstract Syntax Notation)
  - Protobuf
  - Flatbuffer
- Reflection based:
  - Oops (https://bitbucket.org/barczpe/oops)
  - Reflect (https://github.com/simonask/reflect)
- Different requirements e.g.: fast ←→ self descriptive



Oops

Why Oops?

- Simple to use, configurable
- No common base class
- Robust:
  - Automatically handles different class versions
  - Human editable text files
- File driven reading
- File formats (Binary, Text, Yaml)
  - Messages, sections are independent and self descriptive
  - Designed for C++:
    - name, type, value (other file formats have only name-value pair)
    - pointers, smart pointers, optional
    - containers, associative containers (no associative containers in other file formats)

Oops example 1

Add property interface to class A

```
class A
{
public:
    A() = default;
private:
    int i1 = 0;
    std::string s1;
    rOOPS_ADD_PROPERTY_INTERFACE(A)
    {
       rOOPS_PROPERTY(i1);
       rOOPS_PROPERTY(s1);
    }
};
```

Oops example 2

Add property interface to class A

```cpp
class A : public B {
public:
    A() = default;
private:
    int i1 = 0;
    std::string s1;
    void validate(std::uint8_t aVersion) { }
    rOOPS_ADD_PROPERTY_INTERFACE(A) {
       rOOPS_VERSION(1);
       rOOPS_INHERIT(B);
       rOOPS_PROPERTY(i1);
       rOOPS_PROPERTY(s1);
       rOOPS_VALIDATE(&A::validate);
    }
};
```

Oops example

Save and load class A

```
A a(1, "2");
std::stringstream strm;
// Create format object.
rOops::rOopsTextFormat_c frmt(strm);
// Save 'a' to the stream object.
rOops::save(frmt, a, "a");

// Create a parser.
rOops::rOopsTextParser_c parser(strm, "a");
// Load 'a2' from 'strm_txt2'.
A a2;
load(parser, a2);
```

```
a = !A {
 i1 = 1;
 s1 = "2";
};
```

Some interesting feature

- Inheritance, multiple inheritance: `rOOPS_INHERIT`()
- Property can be anything:
  - Template class, smart pointer, optional, enum, chrono, units, etc.
  - Standard containers, e.g.: `std::map<KeyClass, std::shared_ptr<BaseClass>>`
- Give a different name to the property:
  - `rOOPS_PROPERTY(longVariableName_), “l1”;`
- Default value, Required property
- Validate function: `rOOPS_VALIDATE()`
- Version number for class types: `rOOPS_VERSION()`
- Non-invasive version

Limitations

- Must have a default constructor
  - Imagine, that the class need to be constructed before loading.
- Classes cannot have reference members.
- The type descriptor must be manually created for some template types. In the global namespace.

```
using TestDataArray_5 = std::array<rOops::test::Data, 5>;
rOOPS_DECLARE_STL_ARRAY_TYPE_INFO(TestDataArray_5)
rOOPS_DECLARE_STL_LIST_TYPE_INFO(std::vector<Data>)
using IntStringMap = std::multimap<short, std::string>;
rOOPS_DECLARE_ASSOCIATIVE_CONTAINER_TYPE_INFO(IntStringMap)
```

How it works? Type info

- rPropTypeInfo interface class:
  - Type name, type name alias
  - Type id, a hash generated from type name
  - value(), setValue() for converting the object to/from string
  - save() and load() functions for all supported file format
  - Create and destroy functions
- All types has its own Type Info class.
- All Type Info class has a static instance.
- Get type info functions:
  - rPropGetTypeInfo(T*)
  - rPropGetTypeInfoByName(),
  - rPropGetTypeInfoById()

Class hierarchy of the Type Info classes

- rPropTypeInfo
  - rPropTypeInfoBase<T>
    - Integer
    - Float
    - String
  - rPropTypeInfoCompoundTypeBase<T>
    - rPropTypeInfoCompoundAbstract<T>
    - rPropTypeInfoCompoundCreateDestroy<T>
  - rPropTypeInfoSTLContainerBase<T>
    - Array
    - List
    - Set
    - Associative

Base Types

- Everything which is not a container, class or struct.
- The value / save / load functions do not call other such functions.
- Integer and floating-point types are trivial.
- Bool: false/true or 0/1
- Character is written as ‘a’
- Std::string is also a base type and written as “xyz”
- enum – convert enum value to / from string
- Std::chrono duration and time point
  - Smart conversion from different date-time formats
- Custom types, e.g.: complex number written as 3.0+j1.2.

Compound Types
class or struct

- Property Descriptor Table
  - Contains an entry for all properties (data members, parent classes)
- Property Descriptor:
  - Property name, id (hash of name)
  - Reference to the Type Info object
  - Offset of the data member
  - Flags: e.g.: required, read-only
  - Apply pointer for dereferencing pointer members
- Save function iterates over this table and saves every item.
- Load function reads a name – value pair and finds it in this table.

Containers
Associative containers

- Type Info of containers store a Property Descriptor for the elements.
- and for the key, in case of associative containers.
- Knows how to get and set or add an item to the container.
- They also handles pointers on the same way as class member pointers.
- Keys are a bit more difficult.
  - Key is constant or read-only, because changing the key reorders the container.
  - Therefore, key values must be read into a temporary variable,
  - and copy / insert / emplace to the container.
- Oops avoid using temporary variables if possible.

File format Existing file formats

- None of the file formats are designed for C++.
  - Json, Yaml, XML, AmazonIon, etc.
  - ‘:’ as value separator? ‘=‘?
  - Only name value pairs are supported. No place for type.
  - How to handle pointers and associative containers?
  - Type names contains ‘::’ and ‘<‘ ‘>’ characters.
  - These problems must be solved:
    - Introducing format inside the format
    - Extra quotation is required around type names for correct parsing

File format
Oops file formats

Oops text format:

Easy to parse, LL(1) parser is enough

Supports all C++ data structure

Whitespace has no meaning, just separate tokens

It is possible to parse and read as a stream

Oops binary format:

Simple

Independent

It is possible to read as a stream

File format
Oops text format

- Tokens: Numbers (int, float), character (‘c’), string (“xyz”)
- Name value pairs: intValue = 23;
- Class or struct: { m1=1; m2=“xyz”}
- Container: [1, 2, 3]
  - Or: IntList[2] = !sTut_06a::IntList_t [-90, -120];
- Associative container:
  - <“k1”=1, “k2”=2>
- Pointers:
  - IntPtrList[2] = !sTut_06a::IntPtrList_t [&1635937476896, &1635937475808];
  - *1635937476896 = !int32_t 0;

File Format
Oops Binary Format

- The Oops Binary format is designed to be stateless.
- It contains a list of atomic items: atomic data values and meta data.
- Every items have 3 bytes long header:
  - Marker byte: protocol, endianness
  - Control byte for what the item contains: type, name, version
  - Size control byte for describing the size of the item
- Name and Type Name can be written as string or hash value.
- Address of the data can be saved with all items (pointer support)
- Block / list /map begin-end are handled in the control byte.

Future development

- C++17 library support (variant, string view)
- Performance optimization
- Handle escape characters in character and string representations
- Improve documentations

Cereal

Cereal features

- Cereal is similar to boost's serialization library.
- Header only, fast, and minimal.
- Support most of the types in the standard library up to C++11.
- Raw pointers are not supported.
- Supports Binary, Portable Binary, JSON, and XML archives.
- Saving and loading may happen in the same function, called serialize().
  - Its argument decides if it saving (output archive) or loading (input archive).
- Read the archive as the serialize function does.
- No error handling.

Cereal features

Example

```
class A
{
public:
  template <class Archive>
  void serialize(Archive& archive, const uint32_t version)
  {
    archive(CEREAL_NVP(i1));
    if (2 <= version) archive(CEREAL_NVP(s1));
  }
private:
  int i1 = 0;
  std::string s1;
};
CEREAL_REGISTER_TYPE(A)
CEREAL_CLASS_VERSION(A, 2)
```

Use cases

- Configuration files
- Application generator
- Message serialization
- Storing data
- Persistent objects

Use cases
Configuration files

- Challenge:
  - Human readable and editable files
- Robust parser:
  - Missing items, extra items, changing the order of items
  - Meaningful error messages from the parser including line number
  - Whitespace has no meaning, format as you wish
- Not just reading, but also saving the configuration
- Pointers are handled automatically.
- C++ preprocessor for using include statements and macros
- Application generator: build the app by loading objects.

Use cases
Message serialization

- Challenge:
  - Performance, small size of serialized data
  - Handle errors: broken connection, late client
- Binary format:
  - Fast and efficient save() and load() functions
  - Send only data different from the default
- Message independence / stateless stream:
  - The serialized version of the message is independent of the previous content of the stream.
  - Client can start reading the stream any time.
- I/O is the real bottleneck, not the serialization.

Uses cases
Storing data, persistence

- Challenge:
  - How to handle different versions? How to ensure backward compatibility?
- Oops handles the most frequent changes automatically:
  - Adding a new member.
  - Deleting an old member.
- There are support for more complicated cases:
  - Changing the type of a member, or rename a member
  - Read-only properties for reading deprecated properties.
- Pointers are handled automatically in Oops.
- These are all unique for Oops:
  - Different versions and pointers must be handled in the load function manually.

Reflection in the C++ Standard
reflexpr keyword

- https://en.cppreference.com/w/cpp/keyword/reflexpr
- **Usage**
  1. gets the member list of a class type, or the enumerator list of an enum type.
  2. gets the name of type and member.
  3. detects whether a data member is static or constexpr.
  4. detects whether member function is virtual, public, protected or private.
  5. get the *row* and *column* of the source code when the type defines.
- No implementation available yet.

Reflection in the C++ Standard

Proposals

- Static reflection
  - https://isocpp.org/files/papers/n3996.pdf
  - Based on Mirror library 2006-2011
    - http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html
  - Compile time reflection
  - Run-time reflection can be built on top of it.