Serialization of data structures in the C++ programming language using the reflection information

Radosław Sokół*

*Faculty of Electrical Engineering, Silesian University of Technology, Akademicka 10, Gliwice, Poland, e-mail: radoslaw.sokol@polsl.pl

Abstract **In all cases a data structure (or an array of data structures) needs to be serialized or deserialized in the C++ programming language, a developer needs to write his or her own code. If the serialization mechanism needs to accommodate further expansion of the structure or the structure itself contains variable-length fields (especially strings), the required code can quickly grow quite large and may not be reused in further cases. However, using the reflection mechanism described in [1], one can quickly serialize and deserialize any data structure or container with few lines of code.**

Keywords **C++, reflection, serialization, deserialization, generic programming.**

I. Introduction

As for the binary serialization and deserialization of data structures, the C++ programming language provides only the options of performing a straight binary copy of a memory area containing a single data record to or from a file stream, or overloading the stream operators and open-coding one's self serialization mechanism. As the C++ language lacks a reflection mechanism, one cannot perform simple serialization and deserialization of data containers (especially such as lists or deques). Even single entities of a more complex structure (for instance, containing variable-length strings) introduce difficulties and require a developer to write custom-tailored, one-time use subroutines serializing and deserializing data.

However, using an already implemented reflection mechanism [1] one can implement an automatic serialization infrastructure. This paper describes such an implementation and presents ways to further extend its capabilities.

II. Reflection Mechanism

The reflection mechanism described in [1] builds on top of the features of the C++ language to provide the programmer with insight into structures he or she declares. In contrast to the C# and Java languages, C++ has no such feature built in. The reflection mechanism is crucial to implementation of serialization and deserialization, as it lets the programming library to enumerate a structure's fields, query these fields' types and retrieve or set values of these fields without actual compiler support.

III. Serialization Interface

A class that is meant to be serializable and deserializable should implement – besides the *Reflectible* interface described in [1] – the *Serializable* interface defined as a following template [2, 3]:

```
class Externalizable
{
   public:
   virtual ~Externalizable() throw() {}
   virtual void WriteExternal(Stream &Output) const = 0;
   virtual void ReadExternal(Stream &Input) = 0;
};

template <typename StructType>
class Serializable : public Externalizable
{
   public:
   virtual void WriteExternal(Stream &Output) const;
   virtual void ReadExternal(Stream &Input);
   void SetFieldNamesSerialization(const bool s) throw();
   bool FieldNamesSerialization() const throw();
};
```

The interface's methods have the following meaning:

- *WriteExternal*() — serializes the object to the output stream. The *Externalizable* interface defines only the prototype of the method so that its implementations may serialize in any format. The *Serializable* interface template provides a concrete implementation based on the reflection mechanism;
- *ReadExternal*() — deserializes the object from the input stream.
- *SetFieldNamesSerialization*() — enables or disables serialization of a structure's field names. Disabling it saves space and/or throughput but removes the possibility to deserialize structure after it has been refactored.
- *FieldNameSerialization*() — encapsulates the state of field name serialization toggle.

The *Externalizable* and *Serializable* interfaces have been modelled after Java programming language. The lack of the *transient* keyword may be made up by not defining transient fields in the reflection information.

IV. Usage

With the reflection information set up and stream open, serializing a structure only requires using the *WriteExternal*() method:

```
Structure.WriteExternal(Stream);
```

To deserialize a structure, a similar line is needed:

```
Structure.ReadExternal(Stream);
```

With field names serialization mode enabled, the *ReadExternal*() method of the *Serializable* interface verifies whether the serialized data matches the structure being deserialized, as its field names and types can be retrieved from a stream. With this mode disabled, however, there is no such possibility and it is up to the programmer to

ensure that an application does not try to deserialize mismatching structure.

V. PERFORMANCE

While using the serialization infrastructure described in the paper reduces programming time and improves software robustness, it is expected not to reduce performance. The C++ programming language is often being chosen because of performance reasons and wasting the lead in performance over other languages would remove the need to implement a new programming abstraction already present in competing solutions.

A test was performed to verify performance of the serialization mechanism. Three different methods of data serialization were tested:

- simple binary write of a plain-old-data stucture,
- serialization using the method described in this paper, including names of structure's fields,
- as above, but omitting field names.

Serialized data was redirected to three different data sinks:

- null stream (discarding all data being written),
- local file stream,
- network file stream.

The results have been presented in Table I.

TABLE I
SERIALIZATION PERFORMANCE
(MICROSECONDS PER STRUCTURE)

	Simple binary write	*Serialization*	
		including field names	*excluding field names*
Null stream	0.000	0.880	0.560
Local file	1.282	18.908	6.990
Buffered local file	0.482	2.182	1.140
Network file	180.000	6930.000	2462.000
Buffered network file	9.094	18.446	8.874

An additional test of file size was performed to verify whether the serialization mechanism can be more economical than a simple binary write. The results have been presented in Table II.

TABLE II
SERIALIZED DATA SIZE

Serialization method	*Average bytes per structure*
Simple binary write	160
Serialization including field names	308
Serialization excluding field names	145

One can observe that:

1. In all non-buffered cases, serialization adds a noticable overhead. It can be attributed to a greater number of individual write operations per single structure: while in case of a simple binary write there is only one, serialization requires writing each field of a structure separately;
2. In all buffered cases, serialization becomes competitive with simple binary dump of a structure. Omitting field names helps in reducing the overhead. It is worth noticing that buffered serialization without field names is faster than simple unbuffered binary dump, while being more economical.
3. Serialization including field names necesserily increases file size, as every field must be accompanied by several bytes of field name string. In the test case, it nearly doubled the size of a single structure. However, omitting field names in the serialized stream reduces data size by writing strings optimally instead of dumping whole string buffers (along with unused data) into a stream.

The need to omit field names in order to avoid performance drop seems at first to reduce usefulness of the serialization mechanism. However, in real-life applications such huge data amounts generally come from containers of identical structures. A specialized serialization interface for a container may store field names once and then serialize individual structure omitting field names, thus avoiding the performance overhead and not reducing the functionality of the serialization mechanism.

VI. CONCLUSION

The proposed serialization and deserialization infrastructure reduces programming time and improves software quality by allowing a programmer to concentrate on problems instead of implementations. Thanks to concentration of code dealing with stream input and output and data integrity verification, an application may be more robust, and further quality improvements in the serialization code are application-wide instead of being local.

In the same time, using the serialization mechanism—as with many abstract language constructs—does incur performance penalty. It is up to a programmer to decide whether the performance reduction is justified by greater flexibility and robustness of an application. However, omitting field names from the serialized stream and using stream buffering techniques actually improves performance and does not reduce the C++ programming language's superiority in this area over other solutions [4].

However, the presented solution is not without flaws. Its architecture, based on the Java language, is not enough flexible. The Author is already considering removing the need to implement the *Externalizable* or *Serializable* interfaces in own structure or class. Instead, a solution based on the one used in the C# language will be used, with separate serialization classes using the reflection interface and offering different serialization formats (binary, XML, attribute-value and so on). In such case one can choose format best matching the needs of an application.

VII. REFERENCES

[1] Sokół, R.: ”A Reflection Mechanism in the C++ Programming Language”, XXXIV International Conference SPETO, Ustroń, Poland, 2011.

[2] Stroustrup, B.: “The C++ Programming Language”, Addison-Wesley, 1997.

[3] ISO/IEC 14882:2003 Standard, 2003.

[4] Hundt, R.: “Loop Recognition in C++/Java/Go/Scala ”, Google, 2011.