The new C++ serialization library supporting backward and forward compatibility

Michał Breiter, Robert M. Nowak

Institute of Computer Science, Warsaw University of Technology
Warsaw, Poland

ABSTRACT

We describe new programming library *cereal_fwd* supporting serialization (marshalling) with forward and backward compatibility as well as portability between different platforms. The *cereal_fwd* is able to serialize arbitrary set of C++ data structures, including variable length integer encoding, floating number support, string (text support), deep pointer serialization and deserialization, polymorphic pointers and STL collections.

This library supports selected for its space efficiency. This article describes the proposed method, and benchmarking test comparing this library to: *Boost.Serialization*, *Protocol Buffers*, *C++ cereal*.

Keywords: serialization, marshaling, binary archive, Boost.Serialization, C++ cereal, Protocol Buffers

1. INTRODUCTION

Process of converting object state into a stream of bits is serialization or marshalling. The opposite process, called deserialization or demarschalling is reconstruction of object from series of bits. The serialization is used to achieve persistance e.g. save object state in files, or to communicate e.g. send object through the network.

C++ standard library contains stream representation as well as conversions between a binary or a text formats and built-in data types,[1] but there is no support for more advanced constructions or portability between platforms.

The C++ serialization library should solve following technical issues: support big-endian (e.g. MISC family) and little-endian (e.g. Intel x86 family) processor architectures; properly serialize and deserialize pointers and references, i.e. act also for the data pointed to; properly serialize and deserialize shared pointers, i.e. store only one copy of data pointed and properly store recursive data structures using pointers; properly serialize pointers for objects from class hierarchy, especially when multiple-inheritance and virtual inheritance is used; support string serialization as well as standard collections: arrays, vectors, lists, associative memories and sets. Additionally, C++ language does not have support for reflection i.e. inspection of classes and their fields, which makes serialization of user defined types a challenge.

If the new version of software is created, it may be necessary to change an object's structure. For serialization we define two properties connecting with the object structure changes:

- backward compatibility – when the newer version of software is able to read data saved by the older version;
- forward compatibility - when the older version of software is able to read data saved by the newer version.

Backward compatibility may be achieved by storing archive version into stream or making presence of a field optional, therefore the modules can properly serve the archive. Forward compatibility requires ability to skip unknown fields in input data.

Some of the most popular C++ serialization libraries are:

Further author information: (Send correspondence to Robert Nowak)
Robert Nowak: E-mail: robert.nowak@elka.pw.edu.pl

- *Boost.Serialization*.[2] This library, created in 2002, is widely used C++ serialization library that uses only C++03 facilities to make reversible deconstruction of an arbitrary set of C++ data structures possible, where stream of bytes could be binary data, text data and XML. The serialization can be non-intrusive, the classes to be serialized do not need to derive from a specific base class or implement specific member function. *Boost.Serialization* supports serialization of numbers, strings, deep pointer save and restore, classes with inheritance (multiple inheritance), proper restoration of pointers to shared data and STL containers.[3] This library supports backward compatibility adding independent versioning for each class. The drawback is lack of forward compatibility and portability between platforms for binary format.

- *Protocol Buffers*.[4] This library uses an external description of the data structure to generate a code to serialize and to deserialize objects. This solution was created in Google in 2001 for internal use and published in 2008. There is support for: C++, C#, Go, Java and Python. The library support marshalling and demarshalling of signed and unsigned integers with variable length, floating point numbers, fixed length integers, logical values, character strings using ASCII or UTF-8, binary tables and enumerations. *Protocol Buffers* is popular especially in applications exchanging information between modules developed using different programming languages and operating on different platforms. There is full support for backward and forward compatibility.

- *C++ cereal*;[5] created in 2013, similar to *Boost.Serialization*, but drops support for C++ standards earlier than C++11. *C++ cereal* supports backward compatibility and saving into binary, XML and JSON formats. There is support for deep pointer serialization, however only for `std::shared_ptr`, `std::weak_ptr` and `std::unique_ptr`, the raw pointers and references are not supported. The *C++ cereal* can use functions defined for *Boost.Serialization* to serialize and/or deserialize.

There is still place for new solution, because: *Boost.Serialization* and *C++ cereal* does not support forward compatibility, *Protocol Buffers* library needs external description for data structure and separate tools and steps for code generation.

2. THE *cereal_fwd* SERIALIZATION LIBRARY

The *cereal_fwd* was based on *C++ cereal* library,[5] however a lot of the code was modified, as depicted in Tab. 1.

Language	Type	Files	Empty Lines	Comments	Lines
C++ Headers	same	30	0	3425	5124
	modified	10	0	3	30
	added	12	396	593	2308
	removed	0	0	0	0
C++ Sources	same	4	0	899	4275
	modified	33	0	0	5
	added	21	720	343	3269
	removed	0	5	0	0
Sum	same	35	0	4349	9607
	modified	45	0	3	38
	added	41	1168	957	5793
	removed	4	20	23	119

Table 1. Summary of changes between *cereal_fwd* and the base version of *C++ cereal* library.

The main advantage of new *cereal_fwd* serialization library is its forward compatibility in most common cases including:

- adding new fields at the end of serialization code;

- depict type of serialized field as omitted, therefore this field is not deserialized in newer version of software modules; therefore saving space.

Moreover, the binary data format used by *cereal_fwd* is portable between different platforms.

Library was tested on x86_64 bit Linux with: GCC 6.2.1, Clang 3.9.0, GCC 4.8.5 compilers and on x86_64 64 bit Windows with MSVC compiler. The unit tests checked correctness of new *cereal_fwd* features as well as *C++ cereal* code. The testing coverage is summarized in Tab. 2.

Library	Type	Hit	Total	Coverage [%]
cereal_fwd	Lines	2335	2483	94.0
	Functions	31917	34357	92.9
C++ cereal	Lines	1531	1626	94.2
	Functions	17638	18404	95.8

Table 2. Unit test coverage with comparison to base version of *C++ cereal* library.

Apart from unit tests where data loaded was saved by same program instance, cross platform tests were performed. For it Golden Master tests were made. Data was first saved to files on all testing platforms. Then on each platform when tests are run data saved earlier on other platforms is read. Apart from checking incompatibilities between platforms, this kind of tests can help verify that changes made to library don't break compatibility with older versions. These tests were run additionally on 32-bit MIPS big endian platform.

3. BENCHMARIKING

There is no one established method to compare performance of different serialization mechanisms. In other works usage of various testing data can be found. Queirós[6] focused on comparing libraries using JSON format, real data containing weather forecast was used. Sumaray and Makki[7] designed two types representing book and film data specially for conducting tests. Unfortunately generation of fields values was not described. Gligoric et al.[8] proposed fast deserialization method based on code generation for Java. Comparison with serialization from Java Class Library was made using types created specially for this test but also with objects captured during test executions from selected open source projects.

3.1 Benchmarks

Implemented *cereal_fwd* solution was compared with *Boost.Serialization*, *C++ cereal* and *Protocol Buffers*. The compared parameters were: time taken to serialize and deserialize data, size of saved data, allocated dynamic memory and size of compiled application. The binary archives were compared, because of their speed.

Time taken to serialize data was measured using Google Benchmark library.[9] Library allows easy measurement of execution time for specified fragments of code. Tests can be parameterized using set of arguments. Results can be outputted to JSON and CSV format. Time is measured using high precision clock `std::chrono::high_resolution_clock`. To obtain reliable results specified code fragment is run many times. The values showed are the mean time of all executions. Number of iterations is determined dynamically based on results from trial run.

Size of serialized data may be important for mobile and embedded applications, where available memory and storage space is limited. It may also be crucial for data transferred by mobile or low quality networks. Measured value was total size of serialized data. For tests using random values presented is mean value. *Protocol Buffers* add external data description, *C++ cereal* and *Boost.Serialization* do not add any metadata.

For tests we used random values of fields, apart from running many iterations by Google Benchmark. Presented results are mean values from these runs.

Usage of memory allocated on heap was measured using two values. Total number of allocations was measured counting total number of calls to `malloc`, `calloc` and `realloc` function. Second value was maximal size of heap during each test execution. To capture memory usage `memusage` tool from GNU C Library project[10] was used. This tool runs specified program and changes memory management library to it's own one — `libmemusage.so`.

Library tracks all calls to `malloc`, `calloc`, `realloc` and `free` functions and collects data from them. After program finishes it displays, among others, maximal heap size and number of calls to each function.

For conducted tests it was decided to use several types which have possibly different characteristics from each other and should be made of few other types. This choice should allow evaluation of many independent parts of serialization mechanisms.

3.1.1 Numbers serialization

Primitive types were tested with two categories of tests: saving many fields of same type in class and saving them in collections. Distinction was made because tested libraries have different logic for processing primitive types in objects and arrays. For first category of test class `IntegerClass` with 10 field of `std::int32_t` type was used. Using many field should lower influence of overhead needed to save class information. Using to many field could have negative influence on mechanisms having optimizations for classes with relatively low number of fields.

`IntegerClass` was saved as a single object and as a element in array inside `IntegerClassVect` class. For single object case two tests with different value characteristics were performed. First, values were randomized with uniform distribution on whole range of `std::int32_t` type. Next, values were randomized with uniform normal distribution with mean value of 0 and standard deviation of 127. 127 was chosen because it's maximal value which can be represented using one byte for signed numbers. This distribution should simulate usage of default values and enumeration types. Tests with arrays were performed with different number of elements: 8, 64, 512 and 4096. Every test was repeated 100 times for new random numbers.

Operation	Archive	Time	[%]	Heap [kB]	[%]	Aloc.	[%]
read	boost	1556	100	154	100	3.87k	100
	cereal	464	30	152	99	3.37k	87
	this	490	31	152	99	3.17k	82
	proto	433	28	151	98	3.27k	85
write	boost	1371	100	150	100	4.26k	100
	cereal	233	17	148	98	3.16k	74
	this	870	63	149	99	2.96k	70
	proto	253	18	147	98	2.96k	69

Table 3. Benchmark for *Boost.Serialization*, *C++ cereal*, *Protocol Buffers* and *cereal_fwd* (called **this**) for reading and writing `IntegerClass` objects.

Operation	Archive	Time	[%]	Heap [kB]	[%]	Aloc.	[%]
read	boost	1366	100	150	100	4.26k	100
	cereal	234	17	148	98	3.16k	74
	this	847	62	150	99	2.96k	69
	proto	222	16	147	98	2.96k	69
write	boost	1556	100	154	100	3.87k	100
	cereal	468	30	152	99	3.37k	87
	this	407	26	153	99	3.17k	82
	proto	370	24	151	98	3.27k	85

Table 4. Benchmark for *Boost.Serialization*, *C++ cereal*, *Protocol Buffers* and *cereal_fwd* (called **this**) for reading and writing `IntegerClass` object storing one byte numbers.

Build-in numbers were also saved directly in collections, `std::int32_t` and `float` types were saved to `std::vector`, additionally `std::map` with keys and values of type `std::int32_t` was tested.

Handling for map like types may be different from arrays because it's not possible to accesses continuous memory of whole container. Values were drawn randomly using uniform distribution on whole `std::int32_t`

and `float` range. Tests were made with containers of different sizes: 8, 64, 512, 4096 and 8192, each test was repeated 10 times for new random numbers.

The rank of libraries in term of size, heap and allocation numbers for all collection were similar, therefore it is not included into text.

Operation	Archive	Array size	Time	[%]	Heap [kB]	[%]	Aloc.	[%]
read	boost	8	2305	100	151	100	5.06k	100
		512	48411	100	211	100	5.66k	100
		4096	375163	100	699	100	5.96k	100
	cereal	8	987	43	148	99	3.56k	70
		512	49284	102	207	98	4.16k	74
		4096	387411	103	694	99	4.46k	75
	this	8	3194	139	150	99	3.16k	62
		512	174979	361	209	99	3.76k	66
		4096	1395860	372	696	100	4.06k	68
	proto	8	1698	74	148	98	5.16k	102
		512	84198	174	251	119	107.76k	1905
		4096	660874	176	997	143	825.46k	13857
write	boost	8	2924	100	154	100	4.27k	100
		512	83561	100	168	100	4.27k	100
		4096	653808	100	311	100	4.27k	100
	cereal	8	1816	62	152	99	3.67k	86
		512	90864	109	164	98	3.67k	86
		4096	725390	111	307	99	3.67k	86
	this	8	2434	83	153	99	3.37k	79
		512	137895	165	165	98	3.37k	79
		4096	1102520	169	308	99	3.37k	79
	proto	8	1984	068	152	99	4.47k	105
		512	115557	138	209	124	55.47k	1300
		4096	932303	143	610	196	414.17k	9706

Table 5. Benchmark for *Boost.Serialization*, *C++ cereal*, *Protocol Buffers* and *cereal_fwd* (called **this**) for reading and writing arrays of `IntegerClass` objects.

3.1.2 Pointer reading and writing time comparison

Support for pointers was also tested. In first test non-shared *std::unique_ptr* pointers were used. Generated were full tree of N levels. Every node had two pointers, to left and right subtree. Only leafs had empty pointers. In shared pointers test each node had two *std::shared_ptr* pointers to its children and one *std::weak_ptr* pointer to parent node. Generated was also full tree of N levels. In polymorphic pointer test similar data structure was used. For each level of the tree different derived class was used. Type of class depended on level of tree and was used as parameter for templated class *NodeP<N>*. Number of different types used for this tests was directly proportional to height of the tree. Because *Protocol Buffers* library doesn't support pointers or references it wasn't used for pointer tests.

The pointer serialize and deserialize comparison for are depicted in Tab. 6. The deep pointer serialization and deserialization were tested for different pointer types.

3.2 Executable code comparison size

For this test special applications containing minimal logic for reading and writing object to file were created. Single application used only one serialization mechanism. Compared were size of output application which used dynamic version of tested libraries. *Protocol Buffers* requires separate schema definition of composed types for which serialization and deserialization should be supported. Rest of tested libraries store and load native types

Operation	Archive	Tree level	unique pointer Time	unique pointer Heap [kB]	unique pointer Alloc.	polymorphic pointer Time	polymorphic pointer Heap [kB]	polymorphic pointer Alloc.	shared pointer Time	shared pointer Heap [kB]	shared pointer Alloc.
write	boost	0	1523	114	2.34k	1532	114	3.39k	1553	114	2.35k
		1	1930	114	2.35k	1914	114	3.40k	2039	114	2.35k
		5	10541	114	2.44k	21200	114	3.52k	16933	114	2.44k
		9	157293	148	3.88k	386860	192	5.45k	273995	173	3.88k
	cereal	0	312	114	2.34k	326	114	3.39k	322	114	2.34k
		1	432	114	2.34k	456	114	3.39k	573	114	2.34k
		5	1704	114	2.37k	7260	114	3.44k	5980	114	2.41k
		9	22416	114	2.85k	100703	122	3.93k	108032	145	3.37k
	this	0	261	114	2.34k	236	114	3.39k	254	114	2.34k
		1	378	114	2.34k	336	114	3.39k	464	114	2.34k
		5	2797	114	2.37k	7923	114	3.43k	7033	114	2.40k
		9	41630	114	2.85k	115511	121	3.92k	129773	145	3.37k
read	boost	0	1505	114	2.35k	1556	114	3.40k	1605	114	2.35k
		1	1921	114	2.35k	1735	114	3.40k	2212	114	2.36k
		5	7244	114	2.48k	14623	114	3.57k	18382	114	2.55k
		9	92599	160	4.41k	179874	200	5.99k	322077	181	5.44k
	cereal	0	160	114	2.33k	179	114	3.38k	157	114	2.34k
		1	326	114	2.34k	324	114	3.39k	429	114	2.35k
		5	2117	114	2.40k	7595	114	3.48k	7304	114	2.50k
		9	31642	114	3.36k	114985	134	4.46k	143173	161	4.91k
	this	0	583	114	2.33k	599	114	3.38k	629	114	2.33k
		1	842	114	2.34k	755	114	3.38k	1004	114	2.34k
		5	4087	114	2.40k	10439	114	3.46k	12346	114	2.50k
		9	59479	114	3.36k	158190	134	4.44k	218045	162	4.91k

Table 6. Comparison of deep pointers serialization and deserialization for different pointer types

directly. For each test case, types equivalent to native ones were written in *Protocol Buffers* format and used to generate supporting code. In save tests, data was copied from native to generated types. In load tests data was loaded to generated types and copied to native ones. Copying data between generated and native types and creation of objects from both types was included in measured time and memory usage. This approach simulated usage where generated types are used only during serialization and deserialization process.

Test suite	Size [kB] boost	cereal	**this**	proto
IntegerClass, IntegerClassVect	90.86	54.70	94.79	54.80
Maps	70.88	34.66	94.80	107.01
Arrays	74.88	38.64	94.79	54.82
Unique pointers	126.79	70.49	82.60	
Polymorphic pointers	110.82	74.56	94.66	
Shared pointers	134.83	58.50	90.61	

Table 7. Size of generated application for compared libraries. Pointers are not supported by *Protocol Buffers*.

The code sizes of resultant applications are depicted in Tab 7. Protocol Buffers obtained the lowest sizes, except in the case of an associative array. The code for our solution, as expected, results in creation larger application than the base *C++ cereal* archive. It is caused by extended read and write logic.

4. DISCUSSION

Manual creation of code to marshal and demarshal objects is liable to make mistakes and be time-consuming. The new *cereal_fwd* library allows C++ programmer to serialize and deserialize objects supporting forward and backward compatibility. This library is header only, therefore is easy to integrate. Additionally *cereal_fwd* supports portability between platforms.

The more materials is available in the project repository `https://github.com/breiker/cereal_fwd`, where we provide examples of use and source codes as well as all benchmark results.

Future versions of C++ standard may bring (improvements) enhancements which will help improve or make new serialization libraries. Implementation of Reflection Specification[11] may make it possible to support serialization of user defined types without developer having to manually add code describing fields that need to be saved. Metaclasses proposal[12] may enable generation of serialization code without need for separate tools, during compilation of program.

Acknowledgements

This work was supported by Statutory Founds of Institute of Computer Science.

REFERENCES

1. R. Nowak and A. Pająk, *Język C++: mechanizmy, wzorce, biblioteki [C++ Language: mechanisms, design patterns, libraries]*, BTC, Legionowo, 2010. ISBN 978-83-60233-66-5, `http://www.btc.pl/index.php?productID=177835`.
2. Boost Community, "Boost.Serialization." `https://www.boost.org`. accessed 2019-04-15.
3. R. Nowak, "Zapisywanie stanu obiektów. biblioteka boost::serialization," *Software Developer's Journal* (202), pp. 4 – 13, 2011. `https://depot.ceon.pl/handle/123456789/3163`.
4. Google Inc., "Protocol Buffers – Google's data interchange format." `https://github.com/protocolbuffers/protobuf`. accessed 2019-03-24.
5. W. S. Grant and R. Voorhies, "cereal — A C++11 library for serialization.." `http://uscilab.github.io/cereal/`. accessed 2019-02-15.
6. R. Queirós, "JSON on Mobile: is there an efficient parser?," in *Symposium on Languages, Applications and Technologies (SLATE), 3rd*, pp. 93–100, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.
7. A. Sumaray and S. K. Makki, "A comparison of data serialization formats for optimal efficiency on a mobile platform," in *Proceedings of the 6th international conference on ubiquitous information management and communication*, p. 48, ACM, 2012.
8. M. Gligoric, D. Marinov, and S. Kamin, "Codese: Fast deserialization via code generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, *ISSTA '11*, pp. 298–308, ACM, (New York, NY, USA), 2011.
9. "benchmark — A microbenchmark support library." `https://github.com/google/benchmark`. accessed 26.06.2017.
10. "memusage - profile memory usage of a program." `http://man7.org/linux/man-pages/man1/memusage.1.html`. accessed 2018-06-26.
11. D. Sankel, "Working draft, c++ extensions for reflection." `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf`, 2018.
12. H. Sutter, "Metaclasses: Generative c++." `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf`, 2018.