Chapter

Serialization in Object-Oriented Programming Languages

Konrad Grochowski, Michał Breiter and Robert Nowak

Abstract

This chapter depicts the process of converting object state into a format that can be transmitted or stored in currently used object-oriented programming languages. This process is called serialization (marshaling); the opposite is called deserialization (unmarshalling) processes. It is a low-level technique, and several technical issues should be considered like endianness, size of memory representation, representation of numbers, object references, recursive object connections and others. In this chapter we discuss these issues and give them solutions. We also include a short review of tools currently used, and we showed that meeting all requirements is not possible. Finally, we presented a new C++ library that supports forward compatibility.

Keywords: serialization, marshaling, deserialization, unmarshalling, archive, forward compatibility

1. Introduction

Serialization or *marshaling* is the process of converting object state into a format that can be transmitted or stored. The serialization changes the object state into series of bits. The object state could be reconstructed later in the opposite process, called *deserialization* or *unmarshalling*. The reconstructed object is a semantically identical clone to the original object. The object after serialization is called *archive*. Serialization is a low-level technique that violates encapsulation and breaks the opacity of an abstract data type.

Many popular programming languages have serialization support included in the language core or in the standard library. In the optimal situation serialization (and deserialization), the object does not require any additional development and code. In other programming languages, serialization is supported partially, and the usage needs some support in more advanced cases. In particular, C++ standard library contains stream representation as well as conversions between a binary or text streams and built-in data types. However there is no support for more advanced constructions like pointers, references, variants, collections and objects. Developers are required to rely on additional libraries or to manually write serialization code.

Manual creation of code to write and read object is time-consuming and liable to mistakes. This is the reason why libraries supporting serialization are provided. The other reason for use of external tool to serialization is their support to exchange of information between modules developed in different programming languages or executed on systems with different architectures. This functionality requires language-independent description of the data structure. Serialization tools that allow this data exchange are referred to as *portable*.

The portable serialization is not a straightforward process because:

- Different processor architectures (big-endian, little-endian) lead to a different binary representation of numbers and other objects, which potentially hinders portability.

- Objects accessed by reference should be properly restored in terms of inheritance and multi-base inheritance.

- The support of variants (unions in C), where the same memory buffer is used to store different objects.

- Complex structure, where single object could be referenced multiple times by various pointers and references, needs to be restored properly.

- Various object collections should be supported, including lists and dictionaries.

The good serialization support tools give possibility to choose the so-called archive type, i.e. the format used by serialization. *Archive medium* is a name for file or stream. Serialization archive formats can be divided into two main categories: text-based (stream of text characters) and binary format (stream of bytes) [1]. Typical examples of text-based formats include raw text format, JavaScript Object Notation (JSON) and Extensible Markup Language (XML). Binary formats are more implementation-dependent and are not so standardized. The stream of bytes is mostly memory- and time-efficient; therefore the serialized buffer is the smallest and usually fastest to marshall and deserialize; however the buffer is unreadable to developers and most susceptible for portability issues. Text formats are human-readable, which allows developers to perform manual inspections of archives and usually means easier portability, even across languages, but converting objects to text is usually more time-consuming, and memory footprint highly depends on serialized data and object structure. Raw text format is the most memory-efficient text format and is readable for developers if object structure is simple. JSON is a text format that supports tree-like object structures and allows simple validation; XML is also a text format that supports tree-like object structures [2]; moreover it is self-descriptive and allows data validation; however it takes a lot of memory.

As a result *portability* is used in at least two contexts:

- Portability across machines with different architecture but inside the same language or framework (implementation can rely on language-specific solutions, i.e. default character encoding)

- Portability across various languages and frameworks (usually that includes portability across various platforms), which faces various issues and often needs to introduce various constraints for possible serializable structures

During the software development process, it may be necessary to change the object structure being serialized. If the newer version of software is able to read data saved by the older version, the serialization mechanism has *backward compatibility*. If, additionally, the older version of software is able to read data saved by newer version, the serialization mechanism has *forward compatibility*.

Chapter 2 describes in more details common issues faced by serialization tool developer, followed by examples of existing solutions in Chapter 3. Chapter 4 presents proposed extension to one of the existing tools targeting C++ language. Conclusions are included in Chapter 5.

2. Technical issues for serialization

While designing serialization library, developers face various technical issues like different binary representation of numbers, different encoding of letters, object references, and inheritance. Those issues become especially troublesome when trying to create portable archive. Some of the most common issues related to creation of portable binary archives are depicted below together with possible solutions.

2.1 Numbers

There are two main issues with making archive portable between platforms when storing numbers—endianness and size of memory representation.

Two incompatible formats are in common use to represent larger than 1 byte numerical values when stored into memory: a big-endian, where the most significant byte (i.e. byte containing the most significant bit) is stored first (has the lowest address), and little-endian, where the most significant byte is stored last, has the highest address, as depicted in **Figure 1**. Big-endian is the most common format in data networking (e.g. IPv4, IPv6, TCP and UDP are transmitted in big-endian order), and little-endian is popular for microprocessors (Intel x86 and successors are little-endian, but Motorola 68,000 store numbers in big-endian; PowerPC and ARM support both).

Additionally various languages differently define their 'basic integer type'. For example, languages like Java or C# define `int`. as 32-bit variable, disregarding execution platform architecture. C and C++ use the platform-dependent definition of `int`—it has to be at least 16-bit, usually 32-bit on modern architectures, but can be anything bigger. Python (since version 3) can serve as the most extreme example, where built-in type `int`. is defined as *unbounded*. As a result saving types such as C++'s `std::size_t` or `long` directly, without additional size information, may produce data which may not be readable on other platforms.

Some common solutions include number size as part of serialized data or user forced to explicitly state size of data during serialization and deserialization, for example, by using a method named `writeInt16` or by using types like C++'s `std::uint64_t`. Potentially a more portable but possible less-efficient way is based on variable-length integer encoding, as described in the next section as solution for strings and array length encoding.

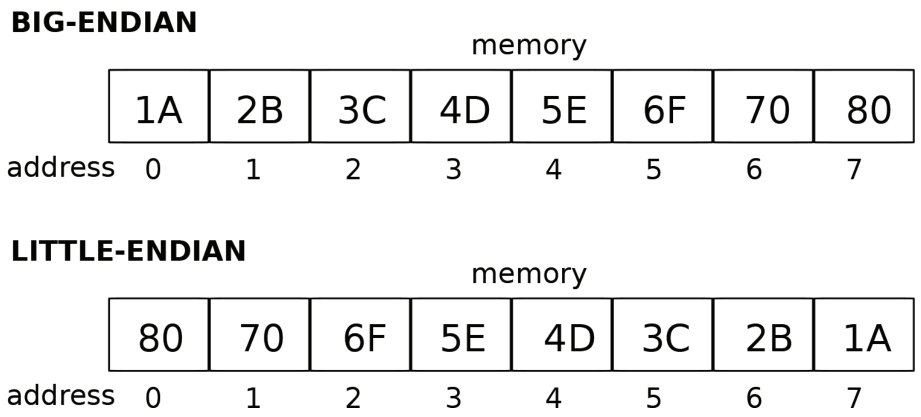

Figure 1.
Representation of `0x1A2B3C4D5E6F7080` *in big-endian and little-endian.*

Floating-point number is more standardized across platforms. The IEEE Standard for Floating-Point Arithmetic (IEEE-754) [3] describes, among others, binary representation of floating-point numbers. It is implemented by most of platforms used today. But that does not necessary solve floating-point numbers' portability issues—for example, the first version of the standard allowed different representations of quiet Not a Number (NaN). For example, Intel x86 and MIPS architecture can use different binary representations for it. Most serialization libraries support only platforms supporting IEEE-754 standard but still need to include their own representation of those NaNs and provide proper conversions.

Handling *long double* type, which is present on some platforms, is especially difficult as it is not standardized, and 64-bit, 80-bit or 128-bit types can be found. Usually portable serialization either does not support that type or defines it by itself, requiring proper conversions to be executed during serialization.

IEEE-754 does not specify endianness used to represent floating-point numbers; in most implementations endianness of floating-point numbers is assumed to be the same as endianness of integers.

2.2 String serialization

The biggest problems with string serialization lie in character encoding and variable-length optimization. Some languages or libraries (C#, Java, C++ Qt) force default encoding of character string in memory (usually from UTF [4] encoding family); others (C, C++) rely on platform or user settings. Serialization tool either forces its chosen encoding, which might lead to both time- and memory-inefficient encoding, or leaves character encoding unchanged and requires users to handle the issue using other means. The latter is usually chosen, as sheer amount of possible combinations of available encoding on various platforms is just enormous.

In most languages character strings can have variable length. This forces serialization process to store length somehow, which raises the question on the type used to store length. Choosing some arbitrary large integer type might be excessive for short strings; choosing too small type might result in problems with serializing huge data chunks. A good solution, which trades some processing time for memory usage, is to use variable-length integer encoding—then, for example, for small arrays, the size of the array is stored using only 1 byte.

The example of variable-length integer encoding is variable-length quantity (VLQ), where 8-bit bytes are used for storing integer and 7-bit types starting from the lowest significant bit are used for coding value, while the most significant bit is used to mark the next byte as part of the encoded integer. The examples are shown in **Figure 2**.

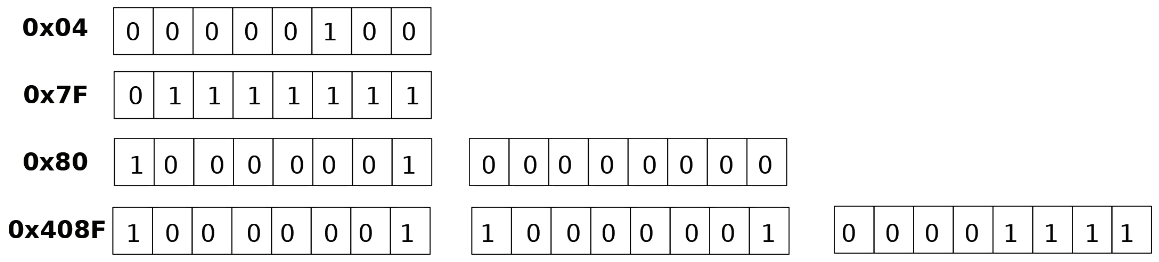

Figure 2.
Unsigned integer numbers variable-length encoded.

2.3 Object representation in memory

Potentially the fastest and easiest way to serialize an object would be to copy contents of memory where that object is stored. Even putting endianness issues aside and focusing on objects composed of only basic types (without pointers or references), this approach just cannot work in a portable way, due to differences in how an object is represented in memory. For efficiency some platforms might enforce specific memory alignment of fields, which introduces padding—empty ‘unused‘bytes between fields. Various platforms can have distinct memory alignments, which in turn can make the same object occupy different amounts of bytes on other systems. Some languages, like C++, give the user partial control over the object’s memory layout, but even those features would not help in creating fully portable object representation.

2.4 Constant members and enumeration values

Serialization is a low-level technique, which violates encapsulation and breaks the opacity of an abstract data type. Object deserialization from this point of view is object creation using stream of bytes representing object state. Therefore the deserialization (unmarshalling) ignores constness, for example, by applying `const_cast` in C++. Users should either avoid serializing objects with constant values or provide proper constructors. Otherwise encapsulation has to be broken by serialization tool.

Popular languages provide ‘enumeration types’—list of constants. Usually enumeration values are serialized using their integer representation. Serialization process has to ensure cohesion of those integer representations. The deserialization usually requires to create enumeration value from its integer representation; therefore, as for the constant, it acts as a low-level technique that breaks the rules of encapsulation.

2.5 Object references

A proper complete serialization should follow all references used in the object. Otherwise deserialized object would not be a semantic copy of its source. This means serializing or deserializing pointer or reference should act on pointed data. In other words serialization should do deep pointer (references) save and restore.

This is simple only when each reference is used only once—when object connections are tree-like. Then serialization could just ‘flatten’ such structure, serializing referenced objects as parts of holder objects. Problem arises, when multiple references point to the same object and uniqueness of referenced object is important for proper semantics of the whole structure.

Consider situation depicted in **Figure 3**, where object B is a ‘shallow copy’ of object A (it points to the same data—object C). In such situations only one copy of data should be saved into the stream, as depicted in **Figure 4**. To achieve that serialization tool must introduce some portable object identifiers and reference tracing. The latter might be difficult, depending on the language and its features used—for example, array of elements in C++, where each of the elements could be referenced by its direct address.

The issue becomes even more difficult in case of recursive object connections, depicted in **Figure 5**. The serialization tool has to detect such structures using reference tracing.

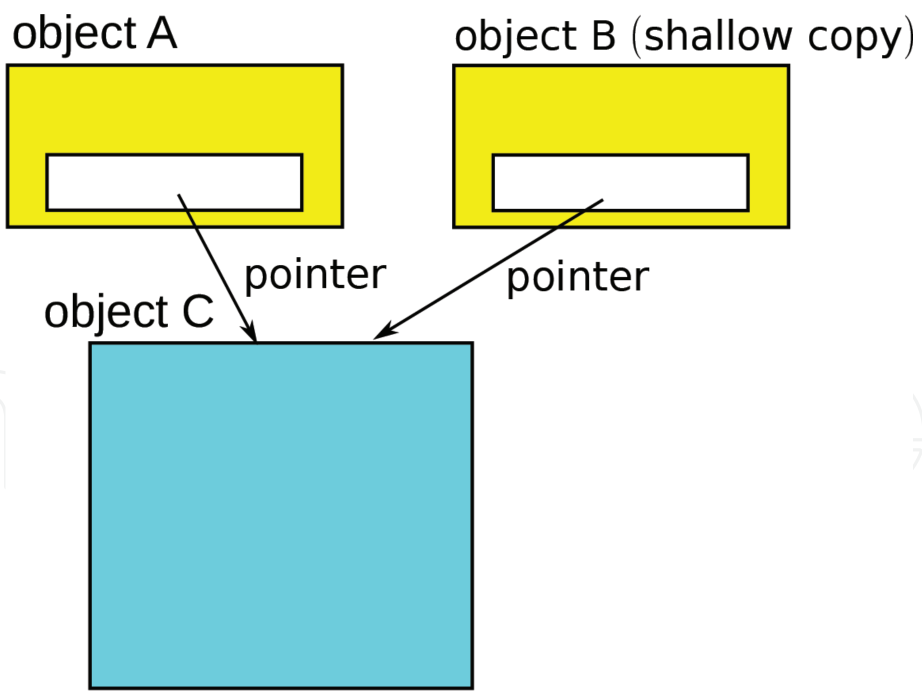

Figure 3.
An example of a shallow copy.

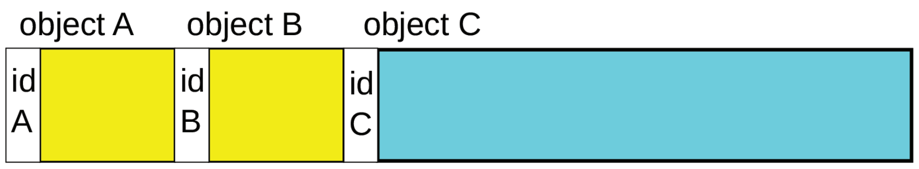

Figure 4.
Shallow copy serialization example.

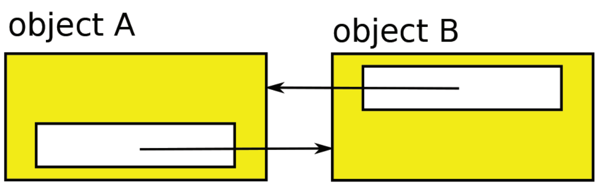

Figure 5.
Recursive object connection.

2.6 Inheritance and polymorphism

The object-oriented languages allow to reference an object by parent class pointer. During marshaling the complete object referenced by pointer should be stored, not the base class used as type of the pointer. This requires access to complete class inheritance hierarchy and runtime type information. The issue here is to retrieve and store unique and portable type identifier (e.g. C++ runtime type information is not portable) and use it to choose proper marshaling and unmarshalling procedures.

Inheritance becomes more troublesome when multiple inheritance is allowed, like in C++, in contrast to languages that permit only multiple interfaces (C#, Java). In such case the so-called diamond inheritance could be created, as depicted in **Figure 6**, where the class inherits from at least two different classes that have the same base class.

If *virtual inheritance* is used, only one instance of base class data is part of the final object; otherwise base class data is present multiple times as part of each class'

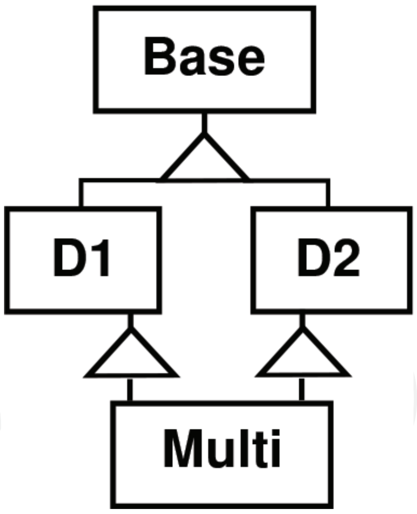

Figure 6.
Diamond inheritance class diagram.

```
class Base {};
class D1 : virtual public Base { };
class D2 : virtual public Base { };
class Multi: public D1, public D2 { };
```

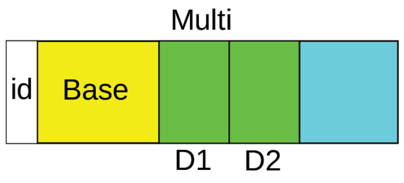

Figure 7.
Virtual diamond inheritance object serialization.

```
class Base {};
class D1 : public Base { };
class D2 : public Base { };
class Multi: public D1, public D2 { };
```

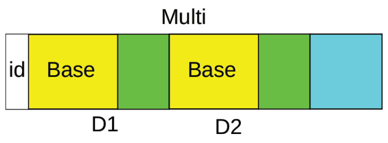

Figure 8.
Non-virtual diamond inheritance object serialization.

parents. The mechanism of serialization must detect which form of inheritance is used and serialize only one copy of the base class in case of virtual inheritance, as depicted in **Figure 7**, or save as many distinct versions of base class data as necessary, as shown in **Figure 8**.

2.7 Collections

It is quite common for serializable structures to contain some sort of data collections, like lists, dictionaries or sets. They do not provide much new issues for serialization process—at least when previously mentioned, serialization problems are solved in the given solution, but they might introduce discrepancies on semantic levels when porting data to different platforms. For example, default standard dictionaries in Java, C# and Python are *hash*-based, but in C++, before C++11 standard, only std::map, an ordered dictionary was available. In some extreme situations, it could lead to data that could not be unmarshalled on C++ side, when dictionary keys did not have natural ordering. On the other hand, data serialized in C++ could lose some of its information on transition to Python or would require usage of non-standard structures. Although most popular languages have similar set of basic collections, some subtle differences might lead to some semantics being 'lost in translation'. Either serialization tool provides its own implementation of common collections for each supported language, which might prove inconvenient for users as it creates impact on existing code of the applications, or serialization library has to introduce some common subset of requirements for collections and somehow verify those requirements before serializing data, which still might introduce some inconvenience for the users, but at least they can still use default collection types in their languages.

2.8 Backward compatibility

The software is constantly modified, and it is useful if the new version of software is able to read data saved by the older version. The backward compatibility can be achieved by storing archive version into stream and keeping deserializing code responsible for handling older versions. That leads to the newest code being the most littered with code for supporting previous versions, but such solution is probably the least memory consuming. Another approach is to use tagged fields, which also help with forward compatibility, as described below.

2.9 Forward compatibility

Forward compatibility allows to use older version of modules when new version of data format is introduced. Supporting forward compatibility requires ability to skip unknown fields in input data. It is usually achieved by adding tags—unique identifiers and type information—to each field. This way during deserialization the software can read only those fields it is aware of. Such approach provides some additional memory footprint, but helps to maintain software in live environment where upgrades are not possible to be performed at once or at all on some instances. Still this solution does not fix all issues—not every type of change in future data format can be made. For example, consider removing field—question arises how older software will interpret missing data, which it still expects. Adding new field should always work, but adding a new type might be troublesome in languages which do not have runtime reflection and type definition capabilities (like C++). Keeping software forward compatible remains mostly the developer's responsibility.

3. Overview of the existing tools

3.1 Java object serialization

One of the probably most known examples of serialization support built-in into language is Java object serialization [5]. Thanks to additional abstraction layer provided by Java virtual machine (JVM), serialization procedures do not need to be concerned with execution environment architecture—memory model, including endianness and object representation, is fixed by virtual machine. That solves some problems with portability of the archive between various real machines. The user only has to mark object using `Serializable` interface and pass object instance to data stream, which uses runtime object reflection to determine object's contents and properly serialize them.

Unfortunately such solution requires all applications involved in marshaling and unmarshalling to be running in JVM, so it is not a real cross-platform portability, but one limited to Java-related ecosystem. Additionally definition of objects has to be shared as parts of the code between applications, with limited support for forward and backward compatibility. Yet out-of-the-box availability and simplicity of use make such solution a good option for the homogeneous systems. Various other platforms implement similar solutions, including. NET `BinaryFormatter` [6] and Python `pickle` [7].

3.2 Google protocol buffers and apache thrift

The common approach for solving platform dependency issues is to introduce *interface description language (IDL)* which allows users to describe data using generalized rules. Such language usually provides limited capabilities, but it makes possible to generate proper code for various platforms. For example, both Google Protocol Buffers [8] and Apache Thrift [9] allow basic numeric types, enumerations, lists, and dictionaries, but no polymorphism or object references. In return those tools can generate serializers and deserializers for significant range of languages, starting with C++, through C# and Java, to Python and JavaScript. Users gain portability in exchange for enforced data structures.

Apache Thrift can serialize objects described with common IDL using various target methods, including human-readable JSON, but for highest efficiency the so-called Compact Protocol should be used, which is similar to the serializer present in Protocol Buffer. Using overhead of field identifier and type encoded before each value, those encoders ensure good forward and backward compatibility. Archive users still need to share part of the code, in the form of the IDL files, but can be implemented in various technologies. Due to built-in forward compatibility support, there is also a smaller chance that change in the IDL would need to propagate to all involved parties than previous solutions.

Some solutions try to remove field identifier overhead from archive and keep forward compatibility support by including whole schema (IDL) inside archive [10, 11]. It can significantly reduce archive size when storing multiple items of the same type. Initial parsing of the schema can introduce some processing overhead, but more importantly such solution might be inconvenient for languages with static typing, where using types created in runtime might be tiresome for developers.

3.3 C++ dedicated solutions

Although C++ is usually supported by cross-language solutions, like Apache Thrift or Google Protocol Buffers, it lacks its own in-language serialization support [12], like Java, C# or Python. Using IDL-based serialization is not always an option for C++ project, as it can be less efficient or too limited than language-specific solution. It is also required to generate all data structures from IDL, so it is not suitable to use in existing project, where serialization should be added to legacy code. As C++ is often used in big legacy projects, the need for language-specific serialization library is justified.

3.3.1 C++ Boost.Serialization

`Boost.Serialization` [13] is a widely used C++ serialization library. This library makes reversible deconstruction of an arbitrary set of C++ data structures possible. It uses only C++03 facilities [14] and therefore could be used in a wide spectrum of C++ compilers, even on the tools that do not support newer C++11 standard [15]. The archive could be binary or raw text or XML. The serialization does not require external record description; additionally the user types to be serialized do not need to derive from a specific base class.

The library supports writing and reading of built-in types (numbers) and most classes from the standard library, like `std::string` and other containers. Enumerations are saved as numeric values. `Boost.Serialization` supports pointer and reference marshaling and demarshaling, i.e. the serialization acts also for the data pointed to. Additionally this library has support for shared pointers (only one copy of data pointed to is saved) and objects with multiple inheritance (also virtual inheritance). Unfortunately, the Boost.Serialization has no forward compatibility. Portability between platforms is achieved only for text formats. In binary format only the data is stored, without additional information that allows the type identification. The numbers are stored as their memory image. The archive is very efficient in terms of size; processes of reading and writing are fast. Unfortunately, there is a lack of portability of binary format.

The backward compatibility is achieved by adding the versioning; therefore the library user can provide different solutions for each version.

To add the serialization for user type, the programmer should implement method serialize, where one of its argument is archive and the second is the version number.

3.3.2 C++ cereal

Similarly to Boost.Serialization, cereal library [16] provides language-specific serialization capabilities. It too makes developer responsible for writing explicit marshaling procedures and supports only backward compatibility. Because it requires serialized structures to use C++11 smart pointers instead of raw pointers and references, library's code can be much simplified, and object tracking becomes easier. Additionally cereal provides support for most of C++ standard library, making it more convenient than Boost.Serialization.

Library contains similar archive types as Boost.Serialization including JSON, XML and binary formats. The major advantage of cereal is the presence of PortableBinary archive, which allows to store data efficiently in binary format in a cross-platform way—Boost.Serialization does not support moving archive across platforms with various endianness, although it is currently being developed by Boost team.

4. cereal_fwd: New serialization library for C++

4.1 Motivation

Currently C++ ecosystem seems to lack efficient and convenient serializing tool supporting portability and forward compatibility. Developers can use some of cross-language tools, like Apache Thrift or Protocol Buffers, but those enforce data types used in application. Users that want to add serialization to existing code base, with already defined C++ classes, are usually left with Boost.Serialization (as part of the most popular Boost library), whose binary archives are not portable. Other solutions, like cereal, do not provide forward compatibility support. The lack of desired tool was a motivation to create new library for serialization of C++ objects.

The main goals for the new C++ library were:

- Support backward and forward compatibility.
- Use minimal size of saved data without hindering ability to evolve structure of serialized data.
- Support streaming for saving and loading operations.
- Minimize allocations during saving and loading process.
- Support portability between different platforms.

- Do not break compatibility for existing archives in library.
- Loading data from unknown source cannot result in undefined behaviour.

4.2 Implementation

New `cereal_fwd` library was based on `cereal` as it already provides some of the required features, and thanks to relying on C++11 language features, it has much simpler implementation than popular `Boost.Serialization`. It might be troublesome for some users, as it requires code to be C++11 compliant, but it helps keep library code simple, and transition towards C++11 should be desired by most existing code's maintainers anyway.

Users of both `cereal` and `cereal_fwd` are required to explicitly list serializable structure's fields in a dedicated method, as shown in **Figure 9**. This is a similar solution to the `Boost.Serialization`, as as of the moment C++ lacks any reflection support, which could help automate the process. Ongoing work [17] suggests that in the future such code could be significantly simplified.

Figure 10 shows how structure can be serialized using selected archive type. In the example `BinaryArchive` is used; `cereal` provides few more archive types, each with the same interface—choosing different archives does not require any changes on structure side. One of the existing archives—`PortableBinary`—provides support for platform portability and was used as a starting point for `cereal_fwd` extension in the form of the `ExtendableBinary` archive. The new archive type is responsible for supporting forward compatibility in `cereal_fwd`.

4.2.1 Numbers

Numbers in `cereal_fwd` are serialized and deserialized similar to `cereal`, but to ensure forward compatibility, all integer type fields are saved with size information attached. This makes it possible to load integers which have different sizes on writing and reading side. The attempt to load integer that cannot fit into type used for loading will result in thrown exception. For space-saving purpose, the size of the saved integer is determined as the minimal number of bytes needed to represent the number being stored.

```
struct Example
{
  std::uint8_t x;
  float y;

  template <class Archive>
  void serialize(Archive& ar)
  {
    ar(x, y);
  }
};
```

Figure 9.
Basic `cereal` *serialization procedure.*

```
int main()
{
  std::ofstream os("out", std::ios::binary);
  cereal::BinaryOutputArchive archive(os);

  Example data;
  archive(data);

  return 0;
}
```

Figure 10.
Serializing structure in `cereal`*.*

4.2.2 Arrays and strings

The size of array or string is saved using variable-length integer encoding—the most significant bit is used to indicate if the next byte is part of stored number. This way for small arrays size is stored using only 1 byte. To speed up serialization and save archive space, all items in the array are assumed to have the same size—size information is stored only once per array.

4.2.3 Polymorphic types

To properly read object stored using polymorphic pointer, identifier of the object's most derived type is needed. In `cereal_fwd` library, as well as in `Boost.Serialization`, for polymorphic pointers a unique type of identifier is saved to stream. Identifier can be any string; by default fully qualified name of the type is used. During reading process identifier is a key in factory which helps in creating correct object and selecting proper deserialization function.

Identifier for specific class is saved only once in stream, for the first occurrence of given type, accompanied with corresponding ordinal number. For every next instance, just the ordinal number is saved. While reading data saved by newer application, it may happen that identifier of polymorphic type will be connected to field which is unknown and will not be normally read. For that reason logic to process every occurrence of type identifier was added.

During code evolution new derived types for existing base classes may be added and saved as polymorphic pointers in archive. In the original version of `cereal` library, the attempt to read unknown polymorphic type results in exception being thrown, consequently stopping reading process. In `cereal_fwd` library an option was added which changes that behaviour. For pointers of unknown type, `nullptr` value is set, and reading process is continued without interruption.

4.2.4 Shared pointers

Existing `cereal` library supports saving several pointers pointing to the same object. In this situation only one copy of the data is saved into the stream. For every other occurrence of the same object, only numerical identifier of previously saved data is stored. For saving process a dictionary of stored pointers and their identifiers is maintained. If address is found in the dictionary, only identifier is saved. For loading process similar mapping between identifiers and shared pointers is maintained. If during loading pointer identifier is already in the map, data is not loaded, and pointer is returned directly from the map.

4.3 Forward compatibility

Forward compatibility was a desired important new feature of `cereal_fwd`, yet implementing it proved to be a demanding task. Some initial assumptions and design decisions were challenged during implementation and are described below.

4.3.1 Adding fields

Supporting possibility for adding fields in newer versions of application should be straightforward—old version of application should just ignore unknown fields. It is true, as long as shared objects are not concerned. Class modification may lead to a new shared pointer field being added, which points to an object which is already used by a different class in the older version of the application. If the first occurrence of shared pointer is saved by field which is present only in the new version and the older version used for reading, reading such shared pointer may be difficult. An example of such situation is depicted in **Figure 11**. Object saved in the archive is A class. In the first version of the application, only one pointer to C class object is saved. In the second version, B::c field pointing to C class object is added. Fields A::c and B::c point to the same object. In the second version, B::c pointer, being part of b field of class A, is saved first. Next, c field of A class is saved. When data is read by the first version of the application, during reading of the B::c, the type of that field is not known, and the whole field could be skipped in basic situation. However, data has to be read in some way to be available for A::c field restoration.

One of the solutions to the described problem is saving stream position of each occurrence of shared pointers and restoring it in case data is needed to read the object by other pointers. This solution was rejected because it would introduce additional requirement on streams supported by the library, to handle rewinding reading position. Such operation is not supported by all streams, i.e. streams formed using network sockets. Another rejected solution was partial interpretation of data and storing it in temporary objects of basic types supported by archive. If the object pointed by shared pointer from unknown field needs to be read by other pointers, instead of using main stream, data would be copied from temporary objects. This

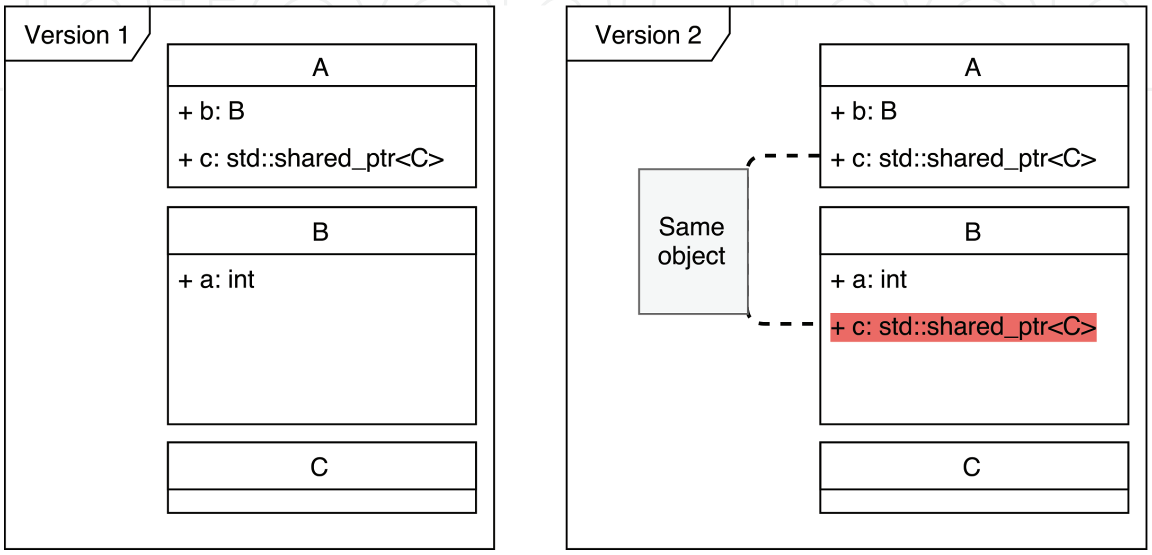

Figure 11.
Example of class evolution showing problem with saving shared pointers.

approach would introduce computational overhead even if no other pointer to the same object was saved.

The chosen solution copies binary data of pointed object of unknown field type to a temporary buffer, by default allocated on heap. In case data needs to be read for earlier omitted pointer, it is read from helper stream created from buffer. Apart from additional stream, stack of reading positions is maintained. It is needed in case omitted object contains additional pointers which were also omitted. An example of this case can be seen in **Figure 12**. The main saved class is `Outer`. In the first version of application, two shared pointers `Outer::q` and `Q::z` are saved. In the second version, pointers are saved in the following order: `Inner::z`, `Inner::q`, `Q::z`, `Outer::q`, `Q::z`. Only single object instances of `Q` and `Z` classes are stored; pointers point to the same objects. When data saved by the second version is read by the first one, data needed by `Outer::q` field can be found in `Inner::q` position. Data for `Q::z` field can be found in `Inner::z` position.

Making copy of data may require a lot of memory. In a worst case size of copied data may be close to the size of all data being read. Such memory allocations may not be acceptable in some applications. Because of that, option to limit maximal size of helper buffer has been added. Trying to use more memory will result in exception being thrown.

4.3.2 Removing fields

Apart from adding new fields, at some point of application evolution, it might be justified to remove no longer needed fields. Saving unnecessary fields results in size and computational overhead. Because in `cereal` order of saving fields is used for field identification, it is forbidden in that library to erase fields from saving and reading methods.

To circumvent this problem, `cereal_fwd` adds ability to change field's type to `OmittedFieldTag`. Using this type indicates position where there was a field but it is no longer used. The older version of application, when trying to read field, which in archive is marked using this tag, will just leave field value in the object unchanged. In most situations it will leave that field with default value assigned in object constructor. It is the responsibility of the developer to erase only such fields

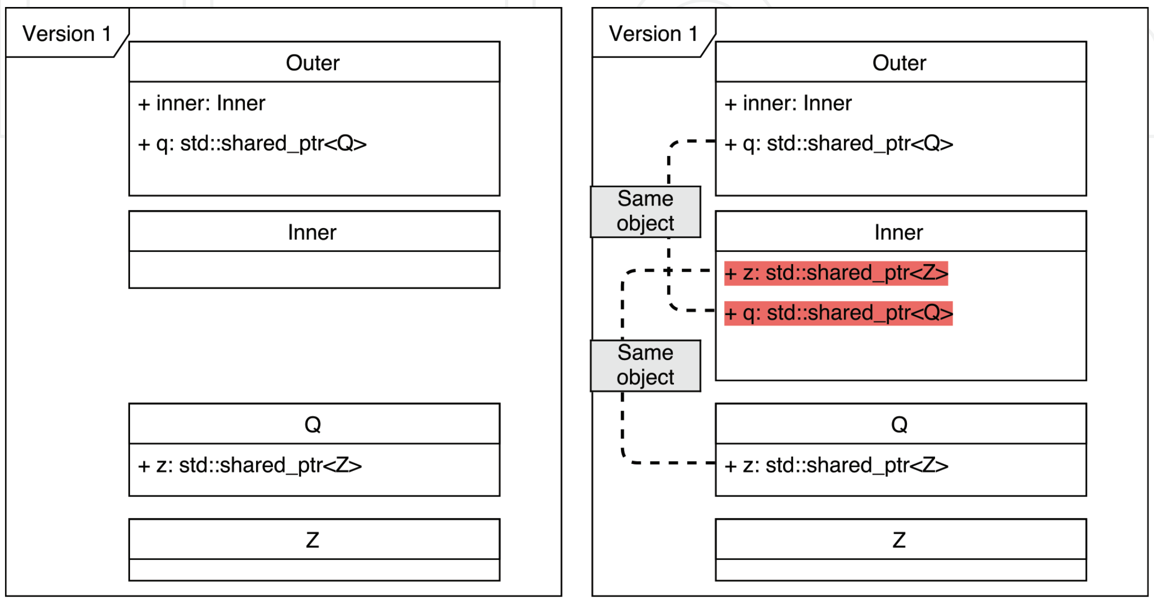

Figure 12.
Example of nested shared pointers.

in the newer application, where default values will still make older versions of the application work correctly. To help the developer create a robust code, method `Archive::wasSerialized` was added. Result of this method indicates if the field being read was really saved or if `OmittedFieldTag` was used in the archive.

4.3.3 `cereal_fwd` *forward compatibility summary*

The `cereal_fwd` supports forward and backward compatibility. The forward compatibility is available, for the following changes:

- Introducing version parameter in `serialize` method, which can be used for conditional serialization code.
- Adding new fields at the end of the object's serialization code; new fields have to be loaded conditionally using class version stored in archive.
- Removing fields from the end of class/struct is permitted.
- Changing the size of integers, as long as stored value is not bigger than target field size; exception is thrown otherwise.
- Renaming structures or classes; the exceptions, caused by storing fully qualified type name in the archive:
- Renamed class cannot be stored using shared pointer.
- Renamed class cannot be saved using pointer to the base class.
- Changing type of serialized field to `OmittedFieldTag`; this change can be done even without changing class version number and introducing conditionals to serialization procedure.
- This leaves some changes in the data structure layout to be still forward incompatible.
- Removing fields without adding `OmittedFieldTag`—trying to load more fields than were saved will result in exception.
- Changing the sign of an integer and loading number that does not fit into a new type, e.g. loading negative number to unsigned integer type.
- Changing the size of floating-point types (e.g. from float into double).
- Changing order in which fields are serialized.
- Adding field for serialization without updating class version and loading it without checking version.
- Adding field for serialization not as the last field.

It is still the developer's responsibility to introduce changes in such a way that archives will be forward compatible.

4.4 Library benchmarks

Performance and memory consumption of newly created `cereal_fwd` was compared against original `cereal`, `Boost.Serialization`, as similar library, and Protocol Buffers as popular non-C++-dedicated solution. The time taken to serialize and deserialize data was comparable; however the fastest for numbers and collection of numbers was Protocol Buffers; the slowest turned out to be `Boost.Serialization`. For pointers the fastest was `cereal` or `cereal_fwd`, the slowest was `Boost.Serialization`, and Protocol Buffers does not support deep pointers serialization.

Usage of memory allocated on heap was measured using total number of allocations (number of calls to memory manager) and maximal size of the heap. All libraries (`Boost.Serialization`, Protocol Buffers, `cereal` and our `cereal_fwd`) had similar usage of the memory when serializing numbers, collections and pointers; all differences were not statistically significant.

Size of created executable was 30% bigger for our `cereal_fwd` than for `cereal` and comparable with `Boost.Serialization`. Protocol Buffers had the smallest code size for serializing numbers, but in the case of collections, code size was the biggest.

The benchmark results are available at project web site.

5. Discussion and conclusion

The support for serialization is required in all currently used programming languages, because there still a growing need to exchange information [18]. The perfect solution, meeting all requirements, does not exist, because the requirements are contradictory:

- The fastest serialization/deserialization process is achieved for binary format, but it is not portable between platforms.

- The most compact is also binary format (without included data description), but such archive is hard to use to exchange information between modules written in different programming languages.

- The text formats, like JSON or XML, are portable and self-descriptive, but serialization/deserialization needs additional data processing, and archive takes significantly more space than the binary one and in result might be slower to transmit if needed.

The programming languages that support reflection have simplified serialize/deserialize process, but other environments needing several technical issues should be resolved, as depicted in Section 2. The existing libraries to serialization require constant development and modernization; because the programming languages are modernized, new standards are implemented, and additional new programming languages are being developed and become used. Our new `cereal_fwd` library addressed the forward compatibility problem for C++ serialization. The library is publicly accessible at `https://github.com/breiker/cereal_fwd` under BSD-like licence.

Acknowledgements

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

Competing interests

The authors declare that they have no competing interests.

Availability of supporting data

Supporting data, including examples of use and source codes, is available in the project repository `https://github.com/breiker/cereal_fwd`.

Author details

Konrad Grochowski, Michał Breiter and Robert Nowak*
Institute of Computer Science, Warsaw University of Technology, Poland

*Address all correspondence to: robert.nowak@pw.edu.pl

References

[1] Sumaray A, Kami Makki S. 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. ACM; 2012. p. 48

[2] Keith W Ballinger, Erik B Christensen, and Stefan H Pharies. Xml serialization and deserialization. US Patent 6,898,604; 2005

[3] IEEE. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-20082008. pp. 1-70

[4] International Organization for Standardization. Unicode. ISO/IEC 10646; 2008

[5] Oracle. Java Object Serialization Specification. 2019. Available from: https://docs.oracle.com/en/java/javase/12/docs/specs/serialization [Accessed: April 15, 2019]

[6] Microsoft. BinaryFormatter Class. 2019. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=netframework-4.7.2 [Accessed: April 15, 2019]

[7] Python Software Foundation. Pickle—Python Object Serialization. 2019. Available from: https://docs.python.org/3/library/pickle.html [Accessed: April 15, 2019]

[8] Google Inc. Protocol Buffers–Google's Data Interchange Format. 2019. Available from: https://github.com/protocolbuffers/protobuf [Accessed: April 15, 2019]

[9] Apache Foundation. Apache Thrift. 2019. Available from: https://thrift.apache.org [Accessed: April 15, 2019]

[10] Apache Foundation. Apache Avro. 2019. Avaliable from: https://thrift.apache.org [Accessed: April 15, 2019]

[11] Huang AS, Olson E, Moore DC. Lcm: Lightweight communications and marshalling. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE; 2010. pp. 4057-4062

[12] Nowak R, Pajak A. C++ Language: Mechanisms, Design Patterns, Libraries. BTC, Legionowo. ISBN: 978-83-60233-66-5; 2010

[13] Boost Community. Boost. Serialization. 2019. Available from: https://www.boost.org [Accessed: April 15, 2019]

[14] International Organization for Standardization. Programming Languages—C++. ISO/IEC 14882:2003, 2003

[15] International Organization for Standardization. Programming Languages—C++. ISO/IEC 14882:2011, 2011.

[16] Shane Grant W, Voorhies R. Cereal. 2019. Available from: http://uscilab.github.io/cereal/ [Accessed: April 15, 2019]

[17] International Organization for Standardization. Working Draft, C++ Extensions for Reflection. 2019. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf [Accessed: April 15, 2019]

[18] Malladi SK, Murphy RF, and Deng W. System and method for processing messages using native data serialization/deserialization in a service-oriented pipeline architecture. US Patent 8,763,008; 2014