Serialization with the `datatools` program library

(datatools version 5.0.0)

F. Mauger <`mauger@lpccaen.in2p3.fr`>

2013-04-22

Abstract

This note explains how to implement class serialization functionnalities using the `datatools` program library serialization interface.

1 Introduction

Serialization means storing and/or loading data to and/or from a storage media. Typical storage media are files that we use everyday to save some data produced by some program A then reload it in memory within program B.
The `datatools` program library proposes some tools to add serialization functionnalities in (mostly) any user classes. It uses the serialization concepts and techniques from the Boost/Serialization library.

Subversion repository:
`https://nemo.lpc-caen.in2p3.fr/svn/datatools/`
in the `doc/Memos/DatatoolsSerializationTutorial` subdirectory
DocDB reference: `NemoDocDB-doc-2003`
References:

- The Boost/Serialization library documentation (`http://www.boost.org/`),
- *Using container objects in* `datatools` (`NemoDocDB-doc-1997`).

2 Basic concepts

2.1 Serialization with the Boost/Serialization library

The Boost/Serialization library by Robert Ramey is based on elegant and powerful concepts and thus provides smart tools to enable serialization of arbitrary objects. The core concept is that the serialization format (ASCII, XML, binary...or any other user format) is distinct from the storage media (streams/files) and thus can be redirect to whatever media the user chooses : files, memory region, containers of bytes...

Practically, for a given class, we must implement the special `serialize` template method that will allow the serialization (input and output) of all (or only part of) data members of the class. It is up to the user to choose what he/she wants to store and load from this specialized serialization method.

The Boost/Serialization library has many features that are out of the scope of this document. However we can mention a few ones:

- support for versioning of serialization for each class,
- support for inherited serialization between mother and daughter classes,
- support for all basic types in C++ (typically using portable *typedefs* from the `boost/cstdint.hpp` header file),
- support for some basic STL containers (vector, list, map, string...),
- support for cross-references through pointers within an serializable object or between different serializable objects,
- support for standard ASCII and XML input and output archives as well as contribution external library implementing binary input and output archives,
- support for *by-pointer* serialization of objects belonging to an inherited class hierarchy.

It should be noticed that all of these features are typically implemented through one single template method per class and need only a few C++ headers that must be included at the beginning of the class header file.
The code, based on templates, is generated at compile time with strict type checking. There is no need for the generation of some complex class dictionary needed by some introspection mechanism as it is done by tools such as `rootcint` within the ROOT framework. In this way, serialization is made very easy using with very limited intrusive code in user's classes and no external tools. Note also that, within an archive, Boost serialization implements a memory tracking algorithm that enables to serialize transient references (pointers) between objects. It is thus possible to save in a file the exact memory layout used by a collection of objects, possibly referencing each others.

2.2 A practical case

The technique to add serialization support to a given class is simple but one must strictly respect a few rules, particularly concerning the headers to include. The sample

1 shows how to add a *serialization* method to the interface of a given class (header file `data.h`). It can be seen that some special declaration instructions have been added in the private part of the class' interface. Particularly, the `serialize` template method is the main interface for Boost/Serialization. Free access to private attributes of the class is garanteed by the declaration of the friend `boost::serialization::access` class.

Here the implementation of the class interface is splitted in two parts:

- the usual implementation of the methods (constructor, getter/setter) of the class is shown in the sample 2 (source file `data.cc`),
- the definition of the `serialize` template method is provided apart and shown in the sample 3 (definition file `data.ipp`).

This implementation splitting approach is recommended for it enables, on user request, to use an alternative definition of the `serialize` template method, without modification of the core class machinery.

The program 1 first serialize (store) a `data` object within a Boost text archive. It then deserialize (load/restore) the object from the Boost text archive. The output file contents is shown in sample 4.

The program 2 behaves like the program 1 but uses Boost XML archives in place of text ones. The output file contents is shown in sample 5.

```
// Standard headers from the STL :
#include <string>
#include <vector>

// Portable integral types (highly recommended):
#include <boost/cstdint.hpp>

// Support for private access to private attribute of a class :
#include <boost/serialization/access.hpp>

// A class that is serializable via Boost archives :
class data
{
public:
  /* The Boost serialization library requests that a serializable
   * class has a default constructor. Here we provide one (note that
   * the compiler will create one if we miss it).
   */
  data ();

  // Getter/Setter methods :
  void set_bval (bool);
  bool is_bval () const;
    // ... more get/set methods could be added...

private:
  // The private attributes of the class :
  bool     _bval_; // A boolean
  int8_t   _cval_; // A signed char (8 bits); implies 'boost/cstdint.hpp'
  int32_t  _ival_; // A 32 bits signed integral
  float    _fval_; // A 32 bits float
  double   _dval_; // A 64 bits float
  std::string _sval_; // A STL string; implies 'boost/serialization/string.hpp'
  std::vector<double> _dvec_; // a STL vector of 64 bits floats;
                              // implies 'boost/serialization/vector.hpp'
private:

  /*******************************
   * Boost/Serialization specific *
   *******************************/

  /* Manage a priviledge access to class' private members
   * by the Boost/Serialization library tools :
   */
  friend class boost::serialization::access;

  /* The main template serialization method :
   *  @arg ar- is of archive type (could be ASCII, XML, binary...)
   *  @arg version_ is the class version number (not used here)
   */
  template<class Archive>
    void serialize (Archive & ar_, const unsigned int version_);
```

Sample 1: The interface of a `data` class enriched with Boost/Serialization support.

```
#include "data.h"

// Implementation of the constructor:
data::data ()
{
  // Initialize members with arbitrary values :
  _bval_ = true;
  _cval_ = '?'; // ASCII code == 63
  _ival_ = 666;
  _fval_ = 3.14159;
  _dval_ = 1. / 3;
  _sval_ = "Hello world !";
  _dvec_.reserve (3); /* pre-allocate the vector's capacity
                       * for memory optimization
                       */
  _dvec_.push_back (_dval_);
  _dvec_.push_back (_dval_ * 2);
  _dvec_.push_back (_dval_ * 3); // only add 3 elements
  return;
}

// Implementation of a setter method :
void data::set_bval (bool bval_)
{
  _bval_ = bval_;
  return;
}

// Implementation of a getter method :
bool data::is_bval () const
{
  return _bval_;
}
```

Sample 2: The implementation of the `data` class. Here we do not put any specific Boost/Serialization related material.

```
2  #ifndef DATA_IPP_
3  #define DATA_IPP_ 1
4
5  #include "data.h"
6
7  // Support for XML 'key-value' based archives (highly recommended):
8  #include <boost/serialization/nvp.hpp>
9
10 // Support Boost serialization of STL string objects :
11 #include <boost/serialization/string.hpp>
12
13 // Support Boost serialization of STL vector containers :
14 #include <boost/serialization/vector.hpp>
15
16 // The implementation of the serialization template method :
17 template<class Archive>
18 void data::serialize (Archive & ar_,
19                       const unsigned int version_)
20 {
21   ar_ & boost::serialization::make_nvp ("bval", _bval_);
22   ar_ & boost::serialization::make_nvp ("cval", _cval_);
23   ar_ & boost::serialization::make_nvp ("ival", _ival_);
24   ar_ & boost::serialization::make_nvp ("fval", _fval_);
25   ar_ & boost::serialization::make_nvp ("dval", _dval_);
26   ar_ & boost::serialization::make_nvp ("sval", _sval_);
27   ar_ & boost::serialization::make_nvp ("dvec", _dvec_);
28   return;
29 }
30 #endif // DATA_IPP_
```

Sample 3: The definition of the `serialize` template method. Note the use of the `&` operator which is the core bi-directionnal I/O archive operator in Boost/Serialization. Also the special *key-value* pair handling construct `boost::serialization::make_nvp("XXX", attribute)` is used to enable XML archive formatting.

```
// Standard C++ file streams :
#include <fstream>

// The text (ASCII) archives from Boost :
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>

// The user's class :
#include "data.h"

// Explicit include of the Boost serialization code for this class :
#include "data.ipp"

int main (void)
{
  /** Serialize a single instance of class 'data' in a
   *  text (ASCII) output archive associated to an output file
   */
  {
    // Declare an output file stream :
    std::ofstream output_stream ("stored_data.txt");

    // Attach a Boost text output archive to the file stream :
    boost::archive::text_oarchive oa (output_stream);

    // Instantiate a 'data' object :
    data my_data;

    // Serialize it (store it in the archive) :
    oa << my_data;

    // Flush the output file stream for safety :
    output_stream.flush ();
  }
  /** Deserialize a single instance of class 'data' in a
   *  text (ASCII) input archive associated to an input file
   */
  {
    // Declare an input file stream :
    std::ifstream input_stream ("stored_data.txt");

    // Attach a Boost text input archive to the file stream :
    boost::archive::text_iarchive ia (input_stream);

    // Instantiate a 'data' object :
    data my_data;

    // Deserialize it (load from the archive) :
    ia >> my_data;
  }
  return 0;
}
```

Program 1: A program that serializes and deserializes an object of `data` class using text archives.

```
1 22 serialization::archive 9 0 0 1 63 666 3.1415901 0.33333333333333331 13 Hello world ! 3 0 0.33333
```

Sample 4: The output file with a Boost text archive produced by program 1. Here the leading `22 serialization::archive 9` string is the Boost archive identifier (archive's header with version 9 in Boost version 1.47.0), following `0 0` are flags to identify the object type and ID within the archive (this could be used in the case of cross-referenced objects), then `1 63 666...` are the serialized class attributes values, i.e. the data.

```
// Standard C++ file streams :
#include <fstream>

// The xml (ASCII) archives from Boost :
#include <boost/archive/xml_oarchive.hpp>
#include <boost/archive/xml_iarchive.hpp>
#include <boost/serialization/nvp.hpp>

// The user's class :
#include "data.h"

// Explicit include of the Boost serialization code for this class :
#include "data.ipp"

int main (void)
{
  /** Serialize a single instance of class 'data' in a
   *  XML (ASCII) output archive associated to an output file
   */
  {
    // Declare an output file stream :
    std::ofstream output_stream ("stored_data.xml");

    // Attach a Boost xml output archive to the file stream :
    boost::archive::xml_oarchive oa (output_stream);

    // Instantiate a 'data' object :
    data my_data;

    // Serialize it (store it in the archive) :
    oa << boost::serialization::make_nvp ("my_data", my_data);

    // Flush the output file stream for safety :
    output_stream.flush ();
  }
  /** Deserialize a single instance of class 'data' in a
   *  XML (ASCII) input archive associated to an input file
   */
  {
    // Declare an input file stream :
    std::ifstream input_stream ("stored_data.xml");

    // Attach a Boost xml input archive to the file stream :
    boost::archive::xml_iarchive ia (input_stream);

    // Instantiate a 'data' object :
    data my_data;

    // Deserialize it (load from the archive) :
    ia >> boost::serialization::make_nvp ("my_data", my_data);
  }
  return 0;
}
```

Program 2: A program that serializes and deserializes an object of `data` class using XML archives.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
<my_data class_id="0" tracking_level="0" version="0">
        <bval>1</bval>
        <cval>63</cval>
        <ival>666</ival>
        <fval>3.1415901</fval>
        <dval>0.33333333333333331</dval>
        <sval>Hello world !</sval>
        <dvec>
                <count>3</count>
                <item_version>0</item_version>
                <item>0.33333333333333331</item>
                <item>0.66666666666666663</item>
                <item>1</item>
        </dvec>
</my_data>
</boost_serialization>

```

Sample 5: The output file with a Boost XML archive produced by program 2. The internal composition of the object can be easily investigated with this human-friendly format which is very useful for debugging purpose. Note the XML format use typical 5 to 10 times more storage than the text format. It is also slower during I/O operations.

2.3 Portability issues

Portability of serialized data is not easy to achieve due to many sources of disagrement between many software and hardware architectures we may use.

Concerning integral types (`bool`, `char`, `int`, `long` and their respective *unsigned* versions), the problem is solved thanks to the use of some type aliases (typedefs) defined in the `boost/cstdint.hpp` header and generalizing the standard `stdint.h` header. It means that the class members of whatever integral types to be serialized must use such aliases as shown in the data sample class above.

For floating values (float, double), it is more complex because the storage of values in the computer RAM depends on the processor architecture (little endian, big endian, hybrid endianness...). More, we must ensure that special values like NaNs (not-a-number) and (plus or minus) infinity (as well as denormalized floating values) are supported. This is tricky.

The Boost/Serialization library implements three kinds of archives (input and output versions) :

- the text archives use an ASCII format is not strictly portable by default because it does not handle non-finite values for floating point numbers, however, for integer types, it is ok, provided that the resources in `boost/cstdint.hpp` are used,
- the XML archives also uses an ASCII format and has the same limitation as the text archives,
- the original Boost binary archives is portable for integer numbers but does not support the (de)serialization of floating point numbers; we do not recommend its usage, which is very limited and not adapted for scientific purpose.

Practically the text and XML archives can be made portable if we use special features from the standard iostream library in order to store and load NaNs and infinite values correctly. The use of the `boost/cstdint.hpp` header implies portability for integer numbers. In order to benefit of the advantages of binary formatted archives (fastness and compactness of storage), the portability issue can be solved using some contributed library from Boost (the *floating point utilities* by John Rade) coupled with special implementation of a portable binary archive by Christian Pfligersdorffer.

The `datatools` library addresses these issues through some special reader/writer classes that are implemented in order to garantee portable serialization.

3 Using serialization within `datatools`

3.1 Reader and writer classes

The `datatools` implements some generic serialization reader and writer classes :

- `datatools::data_writer` : a general purpose serializer class (writer)
- `datatools::data_reader` : a general purpose deserializer class (reader)

These classes are available from the `datatools/io_factory.h` header file.
The storage media consists in a standard file (I/O file stream) which can be compressed (or not) using:

- no compression : no additional suffix,
- GZIP compression : suffix is `".gz"`,
- BZIP2 compression : suffix is `".bz2"`.

The writer and reader classes support serialization of different kinds of archive:

- text archives (ASCII) from the Boost/Serialization library : file extension is `".txt"`,
- XML archives (ASCII too) from the Boost/Serialization library : file extension is `".xml"`,
- portable binary archives using the Boost/Serialization library interface [1] : file extension is `".data"`.

The filename's suffix determines automatically the format and compression to be used. Examples:

- `my_data.txt.gz` : corresponds to a GZIP compressed text archive,
- `my_data.xml` : corresponds to a not compressed XML archive,
- `my_data.data.bz2` : corresponds to a BZIP2 compressed binary archive.

Typical usage uses the simple interfaces from these classes:

- the `data_writer` class uses its template `store` method to serialize a serializable object within a file,
- the `data_reader` class uses its template `load` method to deserialize a serialized object from a file,

In their basics implementation the `store` and `load` method requires :

- a reference to the object to be (de)serialized,
- the object must fulfill the Boost serialization interface, i.e. has a `serialize` template method,
- a special character string, namely the *record tag* (or *serial tag*), must be passed to the store/load methods to associate the serialized object with some unique type identifier,

3.2 Examples

The program 3 shows how to use the writer and reader classes to serialize several object of some serializable class. The output file contents is shown in sample 6.

[1] a contribution by Christian Pfligersdorffer

```
// The datatools reader/writer classes :
#include <datatools/io_factory.h>

// The user's class :
#include "data.h"
#include "data.ipp"

int main (void)
{
  // Serialize some 'data' instances :
  {
    // Instantiate some 'data' objects :
    data d1, d2, d3;

    // Declare a writer :
    datatools::data_writer the_writer ("data_3.txt", datatools::using_multi_archives);

    // Serialize it (store them through the writer) :
    the_writer.store ("data", d1);
    the_writer.store ("data", d2);
    the_writer.store ("data", d3);
  }

  // Deserialize some 'data' instances :
  {
    // Instantiate some 'data' objects :
    data d1, d2, d3;

    // Declare a reader :
    datatools::data_reader the_reader ("data_3.txt", datatools::using_multi_archives);

    // Deserialize them (load them through the reader) :
    the_reader.load ("data", d1);
    the_reader.load ("data", d2);
    the_reader.load ("data", d3);
  }
  return 0;
}
```

Program 3: A program that serializes and deserializes an object of the `data` class using XML archives.

```
22 serialization::archive 9 4 data 0 0 1 63 666 3.1415901 0.33333333333333331
  13 Hello world ! 3 0 0.33333333333333331 0.66666666666666663 1

22 serialization::archive 9 4 data 0 0 1 63 666 3.1415901 0.33333333333333331
  13 Hello world ! 3 0 0.33333333333333331 0.66666666666666663 1

22 serialization::archive 9 4 data 0 0 1 63 666 3.1415901 0.33333333333333331
  13 Hello world ! 3 0 0.33333333333333331 0.66666666666666663 1

```

Sample 6: The output file from a writer object produced by the program 3. Here the writer has been configurated to *encapsulate* each `data` object within its own archive (see the `ds::using_multi_archives` flag at the writer's construction), thus we observe three serialization archive blocks. Note that each block has an archive header (`2 serialization::archive 9`) immediately followed by the *record/serial tag* (the `4 data` tokens serialize the "`data`" string) associated to the object which is streamed just after (`0 0 1 63...`).

3.3 Important remarks

3.3.1 Sequential access

It must be noticed that with the reader and writer classes provided by `datatools`, serialized objects can only be stored in files sequentially and then reloaded in the same way. This is thus not possible to implement random access to serialized objects with this simple technique. Archive files generated through the `datatools::data_writer` may be compared with a magnetic tape. Particularly, if the serialization flow failed somewhere in the middle of the streamed media, the stream is likely to be unusable from the failure point to its end.
Note that the reader class has some method to check if some data are available from the stream. This is shown in program 4. The program serializes two different types of objects in some arbitrary order. Then we assume that we don't know the number of serialized objects at load time and also we don't know their ordering in the serialization stream. The program thus uses a deserialization loop that checks the stored *record tag* associated to the next object from the archive stream; this approach allow to determine/guess which *deserialization driver* has to be used to load the next object from the stream. Of course, one needs at least to known in advance the various types of the serialized objects and provide the associated deserialization drivers.

3.3.2 Serialization strategy with `datatools` reader and writer

A `datatools` writer (and reader) object can be initialized with two different strategies (*modes*) :

- One single archive is associated to the file stream and *all* serialized objects are stored within this unique archive. This leads to a file with the following internal structure :

```
<archive>
  <object #1 record tag>
  <object #1 data>
  <object #2 record tag>
  <object #2 data>
  <object #3 record tag>
  <object #4 data>
  ...
  <object #N record tag>
  <object #N data>
</archive>
```

  This behavior implies the following declaration syntax:

```
1 datatools::data_writer the_writer ("data_3.txt", datatools::using_single_archive);
2 datatools::data_reader the_reader ("data_3.txt", datatools::using_single_archive);
```

Because of some internal functionnalities (*memory tracking*) in the Boost/Serialization library, this mode should be used with care. Particularly, user must not modify any object to be serialized during the serialization process. This is because Boost/Serialization acts like a instant snapshot of the memory so the store or load operations must ne considered as *atomic*.

- One archive is associated to *each* serialized object. This leads to a file with as many archives as serialized objects, with the following internal structure :

```
<archive>
  <object #1 record tag>
  <object #1 data>
</archive>
<archive>
  <object #2 record tag>
  <object #2 data>
</archive>
<archive>
  <object #3 record tag>
  <object #4 data>
</archive>
  ...
<archive>
  <object #N record tag>
  <object #N data>
</archive>
```

This behavior implies the following declaration syntax:

```
1 datatools::data_writer the_writer ("data_3.txt", datatools::using_multi_archives);
2 datatools::data_reader the_reader ("data_3.txt", datatools::using_multi_archives);
```

The use of this mode is recommended, unless the user knows what he/she is doing.

Note that if an archive file has been created with one of these modes, it must be read with the same mode. This is the user's responsability to ensure the proper I/O archive mode.

```
#include <datatools/io_factory.h>   // Reader/writer classes.
#include "data.h"                   // Interface for the 'data' class
#include "data.ipp"                 // ... and associated serialization code
#include <datatools/properties.h>   // Interface for the 'properties' class
#include <datatools/properties.ipp> // ... and associated serialization code.

int main (void)
{
  { // Serialize some objects :
    data d1, d2, d3;
    datatools::properties p1, p2;

    datatools::data_writer the_writer ("data_3.xml", datatools::using_multi_archives);
    // Serialize several objects of different types in arbitrary order :
    the_writer.store ("data", d1);       // The choice of the 'data' and 'properties'
    the_writer.store ("properties", p1); // records is arbitrary. However, it defines
    the_writer.store ("data", d2);       // a convention that must be shared
    the_writer.store ("properties", p2); // at the store and load steps.
    the_writer.store ("data", d3);
  }

  { // Deserialize some objects :
    datatools::data_reader the_reader ("data_3.xml", datatools::using_multi_archives);

    // Deserialization loop :
    while (the_reader.has_record_tag ()) {
      if (the_reader.record_tag_is ("data")) {
        // Deserialization driver for class 'data" :
        std::clog << "Load an object with record='data'." << std::endl;
        data d;
        the_reader.load ("data", d);
      } else if (the_reader.record_tag_is ("properties")) {
        // Deserialization driver for class 'datatools::properties" :
        std::clog << "Load an object with record='datatools::properties'."
                  << std::endl;
        datatools::properties p;
        the_reader.load ("properties", p);
      } else {
        // Error handling for unknown record tags :
        std::cerr << "Error: Unknown record tag '" << the_reader.get_record_tag ()
                  << "' !" << std::endl;
        break;
      }
    }
    std::clog << "Warning: Not more object to be deserialized !" << std::endl;
  }
  return 0;
}
```

Program 4: Serialization and deserialization of two different types of objects in some arbitrary order.

4 The `datatools::i_serializable` interface

We have shown in the previous sections that developpers have to provide the `serialize` template method for each class that is supposed to be serialized through Boost/Serialization. However with datatools serialization tools (reader and writer), a special interface is provided in order to equip a given classes with some dynamically inherited *serializable* trait. On this purpose, the `datatools::i_serializable` abstract class has been implemented (from the `datatools/i_serializable.h` header file).

4.1 The *serializable* tag

This interface class simply enforces a virtual `get_serial_tag` method to be associated to the class. The signature of the method is :

```
1 const std::string & get_serial_tag () const;
```

The aim of this method is to provide an unique character string, i.e. the *serialization tag*, that can identify without ambiguity the type of an object stored in a Boost archive. Using the `datatools` reader and writer classes (see above), it is thus possible, for an object `obj1` of which the class fulfills the `i_serializable` interface, to use the alternative methods :

```
1 ...
2 the_writer.store (obj1); // The get_serial_tag () method is invoked
3                          // internally
4 ...
5 the_reader.load (obj1);  // The get_serial_tag () method is invoked
6                          // and checked internally
7 ...
```

This makes life easier for users because once a *serialization tag* has been fixed while designing a given class and implemented through the `i_serializable` interface, one does not have to remind it.

The samples 7, 8 and 9 illustrates the case of a `sdata` class that inherits the `datatools::i_serializable` interface. The program 5 shows how to use the writer and reader classes to serialize several `sdata` objects.

```
4  #include <string>
5  #include <vector>
6  #include <boost/cstdint.hpp>
7  #include <boost/serialization/access.hpp>
8
9  // Use the 'datatools::i_serializable' interface :
10 #include <datatools/i_serializable.h>
11
12 // A class that is serializable via datatools :
13 class sdata : public datatools::i_serializable
14 {
15 public:
16   sdata ();
17   void set_bval (bool);
18   bool is_bval () const;
19
20 private:
21   bool     _bval_; // A boolean
22   int8_t   _cval_; // A signed char (8 bits); implies 'boost/cstdint.hpp'
23   int32_t  _ival_; // A 32 bits signed integral
24   float    _fval_; // A 32 bits float
25   double   _dval_; // A 64 bits float
26   std::string _sval_; // A STL string; implies 'boost/serialization/string.hpp'
27   std::vector<double> _dvec_; // a STL vector of 64 bits floats;
28                               // implies 'boost/serialization/vector.hpp'
29
30 public:
31
32   /***********************************
33    * datatools/serializable specific *
34    ***********************************/
35
36   static const std::string SERIAL_TAG;
37
38   virtual const std::string & get_serial_tag () const;
39
40 private:
41
42   /*******************************
43    * Boost/Serialization specific *
44    *******************************/
45
46   /* Manage a priviledge access to class' private members
47    * by the Boost/Serialization library tools :
48    */
49   friend class boost::serialization::access;
50
51   /* The main template serialization method :
52    *  @arg ar- is of archive type (could be ASCII, XML, binary...)
53    *  @arg version_ is the class version number (not used here)
54    */
55   template<class Archive>
56     void serialize (Archive & ar_, const unsigned int version_);
57 };
```

Sample 7: The interface of a serializable `sdata` class.

```
#include "sdata.h"

// The 'sdata' class serialization tag :
const std::string sdata::SERIAL_TAG = "sdata";

// The virtual method from the 'datatools::i_serializable'
// interface :
const std::string & sdata::get_serial_tag () const
{
  return sdata::SERIAL_TAG;
}

// Implementation of the constructor:
sdata::sdata ()
{
  // Initialize members with arbitrary values :
  _bval_ = true;
  _cval_ = '?'; // ASCII code == 63
  _ival_ = 666;
  _fval_ = 3.14159;
  _dval_ = 1. / 3;
  _sval_ = "Hello world !";
  _dvec_.reserve (3); /* pre-allocate the vector's capacity
                       * for memory optimization
                       */
  _dvec_.push_back (_dval_);
  _dvec_.push_back (_dval_ * 2);
  _dvec_.push_back (_dval_ * 3); // only add 3 elements
  return;
}

// Implementation of a setter method :
void sdata::set_bval (bool bval_)
{
  _bval_ = bval_;
  return;
}

// Implementation of a getter method :
bool sdata::is_bval () const
{
  return _bval_;
}
```

Sample 8: The implementation of the `sdata` class.

```
#ifndef SDATA_IPP_
#define SDATA_IPP_

#include "sdata.h"

#include <boost/serialization/nvp.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/vector.hpp>

template<class Archive>
void sdata::serialize (Archive & ar_,
                       const unsigned int version_)
{
  // Special serialization code for inherited abstract class :
  ar_ & boost::serialization::make_nvp(
    "datatools__serialization__i_serializable",
    boost::serialization::base_object<datatools::i_serializable >
      (*this));
  // Serialization code for class attributes :
  ar_ & boost::serialization::make_nvp ("bval", _bval_);
  ar_ & boost::serialization::make_nvp ("cval", _cval_);
  ar_ & boost::serialization::make_nvp ("ival", _ival_);
  ar_ & boost::serialization::make_nvp ("fval", _fval_);
  ar_ & boost::serialization::make_nvp ("dval", _dval_);
  ar_ & boost::serialization::make_nvp ("sval", _sval_);
  ar_ & boost::serialization::make_nvp ("dvec", _dvec_);
  return;
}
#endif // SDATA_IPP_
```

Sample 9: The definition of the `serialize` template method for the serializable `sdata` class.

```
#include <datatools/io_factory.h>

#include "sdata.h"
#include "sdata.ipp"

int main (void)
{
  // Serialize some 'sdata' instances :
  {
    sdata d1, d2, d3;

    datatools::data_writer the_writer ("data_4.txt", datatools::using_multi_archives);

    the_writer.store (d1); // No need to pass
    the_writer.store (d2); // a serialization tag
    the_writer.store (d3); // to the 'store' method.
  }

  // Deserialize some 'sdata' instances :
  {
    sdata d1, d2, d3;

    datatools::data_reader the_reader ("data_4.txt", datatools::using_multi_archives);

    the_reader.load (d1); // No need to pass
    the_reader.load (d2); // a serialization tag
    the_reader.load (d3); // to the 'load' method.
  }
  return 0;
}
```

Program 5: A program that serializes and deserializes objects of `data` class using `datatools` writer and reader.

As the syntax needed to implement such functionnalities is rather complex, a bunch of useful pre-processor macros are available. The samples 10, 11 and 12 illustrates the use of these macros to define the serializable `sdata` class.

```
12 class sdata : DATATOOLS_SERIALIZABLE_CLASS  // Shortcut macro
13 {
14 public:
15   sdata ();
16   void set_bval (bool);
17   bool is_bval () const;
18
19 private:
20   bool     _bval_;
21   int8_t   _cval_;
22   int32_t  _ival_;
23   float    _fval_;
24   double   _dval_;
25   std::string _sval_;
26   std::vector<double> _dvec_;
27
28   // Shortcut macro for serializable class interface :
29   DATATOOLS_SERIALIZATION_DECLARATION();
30
31 };
```

Sample 10: The interface of a serializable `sdata` class.

```
3 // Shortcut macro to implement the 'serialization tag' stuff :
4 DATATOOLS_SERIALIZATION_SERIAL_TAG_IMPLEMENTATION (sdata, "sdata")
```

Sample 11: The implementation of the serialization features of the `sdata` class.

```
#include <boost/serialization/string.hpp>
#include <boost/serialization/vector.hpp>

template<class Archive>
void sdata::serialize (Archive & ar_,
                       const unsigned int version_)
{
  // Shortcut macro to implement serialization code for
  // the inherited abstract class :
  ar_ & DATATOOLS_SERIALIZATION_I_SERIALIZABLE_BASE_OBJECT_NVP;
  // Serialization of class attributes :
  ar_ & boost::serialization::make_nvp ("bval", _bval_);
  ar_ & boost::serialization::make_nvp ("cval", _cval_);
  ar_ & boost::serialization::make_nvp ("ival", _ival_);
  ar_ & boost::serialization::make_nvp ("fval", _fval_);
  ar_ & boost::serialization::make_nvp ("dval", _dval_);
  ar_ & boost::serialization::make_nvp ("sval", _sval_);
  ar_ & boost::serialization::make_nvp ("dvec", _dvec_);
  return;
}

#endif // SDATA2_IPP_
```

Sample 12: The definition of the `serialize` template method for the serializable `sdata` class (this version uses a macro).

5 Serialization through pointers to an abstract mother class

The Boost/Serialization library provides a mechanism to automatically serialize objects derived from a polymorphic mother class. In this case serialization can be performed through pointers to the base class.

This system relies on some useful macros to automate a complex mechanism to register the mother/daughter class relationships between classes and determine the actual derived type of a serialized object.

Practically, the `datatools::i_serializable` interface behaves like such a mother class. Thus, each class that inherits from it can be (de)serialized through this mechanism.

The program samples 13, 14, 15 and 16 illustrate the case of serializable objects of classes `sdata1` and `sdata2` derived from `datatools::i_serializable`. An array of pointers to such *serializable* objects is filled with such randomly generated objects. The array is then serialized through a `datatools` writer that uses a text archive file. In a second step, the array is deserialized from the file through a `datatools` reader.

data_6.cxx

```
#include <datatools/i_serializable.h>
#include <boost/serialization/nvp.hpp>

class sdata1 : DATATOOLS_SERIALIZABLE_CLASS  // Shortcut macro
{
private:
  bool _flag_;
public:
  sdata1 () ;
  DATATOOLS_SERIALIZATION_DECLARATION(); // Shortcut macro
};

// Declaration macro of the class exporting code (GUID) :
#include <boost/serialization/export.hpp>
BOOST_CLASS_EXPORT_KEY2(sdata1, "sdata1")

class sdata2 : DATATOOLS_SERIALIZABLE_CLASS  // Shortcut macro
{
private:
  double _value_;
public:
  sdata2 () ;
  DATATOOLS_SERIALIZATION_DECLARATION(); // Shortcut macro
};

// Declaration macro of the class exporting code (GUID) :
BOOST_CLASS_EXPORT_KEY2(sdata2, "sdata2")
```

Sample 13: The interface of classes `sdata1` and `sdata2`.

When one creates some new serializable `foo` class with this technique, it is strongly recommended to split the interface (header file `foo.h` as in sample 13), the implementa-

data_6.cxx

```
30 sdata1::sdata1 () : _flag_ (true) {}
31
32 // Shortcut macro :
33 DATATOOLS_SERIALIZATION_SERIAL_TAG_IMPLEMENTATION (sdata1, "sdata1")
34
35 sdata2::sdata2 () : _value_ (0.0) {}
36
37 // Shortcut macro :
38 DATATOOLS_SERIALIZATION_SERIAL_TAG_IMPLEMENTATION (sdata2, "sdata2")
```

Sample 14: The implementation code for classes `sdata1` and `sdata2`.

tion (source file `foo.cc` as in sample 14) and the code specific to serialization (`foo.ipp` as in sample 15). The good approach is :

- to build an object file from the implementation file :

  `foo.h` +`foo.cc` → `foo.o`

- to build an additional object file from the associated serialization code only :

  `foo.h` + `foo.ipp` + INSTANTIATION/EXPORT macros → `foo_bio.o`

  where the "`_bio`" suffix refers to the Boost I/O system.

This allows:

- to link easily some executable with both object files `foo.o` and `foo_bio.o`,
- but also, if we don't need the serialization functionnalities, to link only with the `foo.o` object file,
- and finally, to link with `foo.o` and an alternative serialization object file (`alt_foo_bio.o`) that must be provided in place of the default `foo_bio.o` if this last one does not suit the serialization requirements for a specific application.

data_6.cxx

```
41 template<class Archive>
42 void sdata1::serialize (Archive & ar_, const unsigned int version_)
43 {
44   ar_ & DATATOOLS_SERIALIZATION_I_SERIALIZABLE_BASE_OBJECT_NVP;
45   ar_ & boost::serialization::make_nvp ("flag", _flag_);
46   return;
47 }
48
49 template<class Archive>
50 void sdata2::serialize (Archive & ar_, const unsigned int version_)
51 {
52   ar_ & DATATOOLS_SERIALIZATION_I_SERIALIZABLE_BASE_OBJECT_NVP;
53   ar_ & boost::serialization::make_nvp ("value", _value_);
54   return;
55 }
56
57 /** Implementation of the classes' exporting code and automatic
58  *  instantiation of template code for all Boost archives
59  *  used by the datatools library. This code segment must be
60  *  invoked from the main program or pre-compiled within a DLL.
61  */
62 #include <datatools/archives_instantiation.h>
63 DATATOOLS_SERIALIZATION_CLASS_SERIALIZE_INSTANTIATE_ALL(sdata1)
64 BOOST_CLASS_EXPORT_IMPLEMENT(sdata1)
65
66 DATATOOLS_SERIALIZATION_CLASS_SERIALIZE_INSTANTIATE_ALL(sdata2)
67 BOOST_CLASS_EXPORT_IMPLEMENT(sdata2)
```

Sample 15: Definition and automated instantiation of serialization and exporting code.

data_6.cxx

```
#include <cstdlib>
#include <datatools/io_factory.h>
#include <boost/serialization/vector.hpp>
int main (void)
{
  srand48 (314159); // Initialize the drand48 PRNG.
  typedef std::vector<datatools::i_serializable *> data_array_type; // Useful typedef

  // Serialize some mixed 'sdata1' and 'sdata2' instances :
  {
    data_array_type data;
    // Populate the array of data :
    std::cout << "Data to be stored : " << std::endl;
    for (int i = 0; i < 10; ++i) {
        datatools::i_serializable * a_data_ptr;
        // Randomize (50%/50%) the type of data to be stored in the array :
        if (drand48 () < 0.5) a_data_ptr = new sdata1;
        else a_data_ptr = new sdata2;
        data.push_back (a_data_ptr);
        std::cout << "Serial tag is '" << data.back ()->get_serial_tag ()
                  << "'" << std::endl;
      }
    std::cout << std::endl;
    datatools::data_writer the_writer ("data_6.txt", datatools::using_multi_archives);

    // Serialize the array of data pointers :
    the_writer.store ("array", data);
    for (int i = 0; i < data.size (); ++i) delete data[i];
  }

  // Deserialize some mixed 'sdata1' and 'sdata2' instances :
  {
    data_array_type data;
    datatools::data_reader the_reader ("data_6.txt", datatools::using_multi_archives);

    // Deserialize the array of data pointers :
    the_reader.load ("array", data);

    std::cout << "Loaded data : " << std::endl;
    for (int i = 0; i < data.size (); ++i) {
        std::cout << "Serial tag is '" << data[i]->get_serial_tag ()
                  << "'" << std::endl;
      }
    for (int i = 0; i < data.size (); ++i) delete data[i];
  }
  return 0;
}
```

Sample 16: Main program.

6 Conclusion

The `datatools` library implements serialization functionnalities based on the Boost/Serialization library. Most of the code is handled through templates and high-level preprocessing macros are provided to ease the implementation of serialization code in users' applications.

An interface for *serializable* object has been designed. The container classes provided by `datatools` fulfill this interface. Particularly, the `datatools::things` generic container can store any of these *serializable* objects.

In this note, some recommendations have been done in order to organize the implementation of the code : splitting the interface, implementation and serialization layers. This allows to use a clear and standard procedure and makes the serialization functionnalities optionnal, versatile and extensible for all new classes that enter a project. The integration with the high-level generic reader and writer classes provided by `datatools` is thus straightforward.

Three Boost archive formats are available from `datatools` in both input and output versions : ASCII text, XML and binary. All are made portable and can handle all integral types up the 64-bits as well as floating point numbers in single and double precision (IEEE754) including non-finite values (NaNs, $\pm\infty$).

The Boost/Serialization library has many features and functionnalities (class versionning, memory tracking, load/save code splitting...) that are out of the scope of this document and thus not covered here. The reader is invited to read the online documentation at:
`http://www.boost.org/doc/libs/1_48_0/libs/serialization/doc/index.html`.