Boost.Serialization: A Hands-On Tutorial

Sohail Somani
SSCI

sohail@taggedtype.net

May 6, 2008

About this presentation

- Boost.Serialization
- Basic topics
- Practical considerations
- Some advanced topics
- Hands-on exercises throughout

Goals

- Framework design and implementation
- Practical considerations/limitations

About me

- Independent Software Developer

About me

- Independent Software Developer
- Expertise: C++, Boost, multi-platform development

About me

- Independent Software Developer
- Expertise: C++, Boost, multi-platform development
- Experience: Compilers, Grid Computing, 3-D Graphics, Finance, Enterprise & Consumer software

About me

- Independent Software Developer
- Expertise: C++, Boost, multi-platform development
- Experience: Compilers, Grid Computing, 3-D Graphics, Finance, Enterprise & Consumer software
- Love interesting or technically challenging projects (especially if involving any of the above!)

About you

- C++ programmer who does not fear templates (or do a very good job of hiding it!)

About you

- C++ programmer who does not fear templates (or do a very good job of hiding it!)
- Made peace with the command-line (or do a very good job of hiding it!)

About you

- C++ programmer who does not fear templates (or do a very good job of hiding it!)
- Made peace with the command-line (or do a very good job of hiding it!)
- Well-behaved and ask lots of questions

About you

- C++ programmer who does not fear templates (or do a very good job of hiding it!)
- Made peace with the command-line (or do a very good job of hiding it!)
- Well-behaved and ask lots of questions
- Have a laptop with a working C++ compiler

Why?

- Data marshalling for IPC
- Data persistence
- Distributing objects
- Ad-hoc or proprietary file formats
- Template instantiation repository!

Features

- "Reversible deconstruction of an arbitrary set of C++ data structures to a sequence of bytes" - Serialization docs
- Main features:
  - ANSI C++ $\rightarrow$ multiple supported platforms
  - Class versioning
  - Object-graph serialization
  - Serialization for all standard C++ types
  - Serialization for your own datatypes (including polymorphic types)
  - Non-intrusive serialization of third-party types
  - Multiple outputs for easy debugging
  - A non-trivial serialization framework without a frameworky feel

History

- Initial version submitted in February 2002 → Rejected
- Second review April 2004 → Accepted
- Included in 1.32

Single function

```
1  class ChildA : public BaseObject
2  {
3  public:
4    virtual // Single function (a la MFC)
5    void serialize(Archive & ar)
6    {
7      if(ar.is_loading())
8      {
9        ar >> m_data;
10     }
11     else
12     {
13       ar << m_data;
14     }
15   }
16 private:
17   int m_data;
18 };
```

Two functions

```
1  class ChildB : public BaseObject
2  {
3  public:
4    // Two symmetric functions - sometimes
5    // implemented using iostreams
6    virtual void serialize(OStream & os)
7    {
8      os << m_data;
9    }
10   virtual void deserialize(IStream & is)
11   {
12     is >> m_data;
13   }
14
15 private:
16   int m_data;
17 };
```

Problems with typical implementations

- Code triplication
- Requires common base class + virtual functions
- No clear way to serialize third-party classes
- No versioning

Boost.Serialization in a nutshell

```
1  // No base class required!
2  class MyClass
3  {
4  public:
5    // Single function!
6    template<typename Archive>
7    void serialize(Archive & ar)
8    {
9      // No branches!
10     ar & m_data;
11   }
12 private:
13   int m_data;
14 };
```

Archive concept

- Archives serialize to/from some Archive-specific format
- `Ar` is a type modelling the Archive Concept
- `ar` is an instance of type `Ar`
- `T` is a serializable type
- `x` is an instance of type `T`
- `Ar::is_loading` - one of

  `boost::mpl::bool_<true>, boost::mpl::bool_<false>`
- `Ar::is_saving` - one of

  `boost::mpl::bool_<true>, boost::mpl::bool_<false>`
- `ar.register_type<T>()` - Append information about type `T` to archive (aka class registration)
- `ar.register_type(x)` - Same as above but provided for convenience on non-conforming compilers
- `ar.library_version()` - Returns an unsigned integer containing current library version. Incremented when changes could cause the serialization of some type to be changed.

Archive concept

- Archives serialize to/from some Archive-specific format
- `T` is a serializable type
- `ar.register_type<T>()` - Append information about type `T` to archive (aka class registration)

Saving Archive concept

- A Saving Archive is a refinement of the Archive concept
- `SA` is a type modelling the Saving Archive Concept
- `sa` is an instance of type `SA`
- `SA::is_loading` - `boost::mpl::bool_<false>`
- `SA::is_saving` - `boost::mpl::bool_<true>`
- `sa << x` - Append the value of `x` in an Archive-specific manner
- `sa & x` - Must perform exactly the same operation as `sa << x`
- `sa.save_binary(u,count)` - Appends `size_t(count)` bytes found at address `u`

Saving Archive concept

- `sa << x` - Append the value of `x` in an Archive-specific manner
- `sa & x` - Must perform exactly the same operation as `sa << x`

Loading Archive concept

- A Loading Archive is a refinement of the Archive concept
- `LA` is a type modelling the Loading Archive Concept
- `la` is an instance of type `LA`
- `LA::is_loading` - `boost::mpl::bool_<true>`
- `LA::is_saving` - `boost::mpl::bool_<false>`
- `la >> x` - Set `x` to a value retrieved from `la`
- `la & x` - Must perform exactly the same operation as `la >> x`
- `la.load_binary(u,count)` - Retrieves from `la`, `size_t(count)` bytes and stores them at `u`
- `la.reset_object_address(v,u)` - Notify `la` that the object originally at address `u` has been moved to address `v`
- `la.delete_created_pointers()` - Avoid memory leaks that may occur if pointers are being loaded and an exception occurs

Loading Archive concept

- `la >> x` - Set `x` to a value retrieved from `la`
- `la & x` - Must perform exactly the same operation as `la >> x`

Archive models

- Archives provided:
  - `boost::archive::xml_w?[io]archive`
  - `boost::archive::text_w?[io]archive`
  - `boost::archive::binary_w?[io]archive`
  - `boost::archive::polymorphic_w?[io]archive`

Note about Archives

Archives are not streams!

Serializable types

A type `T` is Serializable if:

- It is a primitive type - types that can be serialized by the archives themselves. All built-ins are primitives. `std::string`'s are also considered primitive.
- It is a class type and one of the following has been declared:
  - A class member function `serialize`
  - A global function `serialize`
- It is a pointer to a Serializable trackable type
- It is a reference to a Serializable type
- It is a native C++ array of a Serializable type

Making a class serializable

```
#include <iostream>
#include <string>
struct hello_phrase
{
  explicit hello_phrase(std::string const &
                        whom="World"): m_whom(whom){}
  void say_it(std::ostream & s){ /*... */}

   std::string m_whom;
};
```

Making a class serializable

```
#include <iostream>
#include <string>
struct hello_phrase
{
  explicit hello_phrase(std::string const &
                        whom="World"): m_whom(whom){}
  void say_it(std::ostream & s){ /*... */}
   template<typename Archive> //(1)
   void serialize(Archive & ar, const unsigned int)
   {

   }
   std::string m_whom;
};
```

Making a class serializable

```
#include <iostream>
#include <string>
struct hello_phrase
{
  explicit hello_phrase(std::string const &
                        whom="World"): m_whom(whom){}
  void say_it(std::ostream & s){ /*... */}
   template<typename Archive> //(1)
   void serialize(Archive & ar, const unsigned int)
   {
      ar & m_whom; //(2)
   }
   std::string m_whom;
};
```

Serializing an object - Archives

```
1  #ifndef INCLUDED_ARCHIVE_HPP
2  #define INCLUDED_ARCHIVE_HPP
3
4  #include "boost/archive/text_oarchive.hpp"
5  #include "boost/archive/text_iarchive.hpp"
6
7  typedef boost::archive::text_oarchive oarchive_t;
8  typedef boost::archive::text_iarchive iarchive_t;
9
10 #endif
```

Serializing an object - using C++ output streams

```
1  #include "hello_phrase.hpp"
2  #include "archive.hpp"
3  #include <sstream>
4
5  int main()
6  {
7    hello_phrase hp("Sohail");
8    // Archives load to/from streams
9    std::ostringstream os;
10   {
11     oarchive_t oa(os);
12     oa << hp; // or oa & hp
13   }
14   /* next slide */
15 }
```

Deserializing an object - using C++ input streams

```
1  #include "hello_phrase.hpp"
2  #include "archive.hpp"
3  #include <sstream>
4
5  int main()
6  {
7    /* previous slide */
8
9    // Default constructed
10   hello_phrase hp1;
11   {
12     std::istringstream is(os.str());
13     iarchive_t ia(is);
14     ia >> hp1; // or ia & hp1
15   }
16
17   std::cout << "Original: " << hp << std::endl;
18   std::cout << "Deserialized: " << hp1 << std::endl;
19 }
```

Output - text archive

```
22 serialization::archive 4 0 0 6 Sohail
```

Output - binary archive

Hexdump

```
16 00 00 00 73 65 72 69  61 6c 69 7a 61 74 69 6f
6e 3a 3a 61 72 63 68 69  76 65 04 04 04 04 08 01
00 00 00 00 00 06 00 00  00 53 6f 68 61 69 6c 0a

|....serializatio|
|n::archive......|
|.........Sohail.|
```

Output - XML archive

- Oops, doesn't compile

```
error: no matching function for call to
assertion_failed(mpl_::failed************
boost::serialization::is_wrapper<hello_phrase>::
************)
```

- AHHHHHH!

Use of compile-time asserts

The failing section of code (in boost/archive/basic_xml_iarchive.hpp):

```
1  // Anything not an attribute and not a
2  // name-value pair is an
3  // should be trapped here.
4  template<class T>
5  void load_override(T & t, BOOST_PFTO int)
6  {
7    // If your program fails to compile here,
8    // its most likely due to
9    // not specifying an nvp wrapper around the
10   // variable to
11   // be serialized.
12   BOOST_MPL_ASSERT((serialization::is_wrapper<T>));
13   ...
```

Name-value pairs

- XML archive requires a name for each value
- Most simple name to use is the variable name
- `boost::serialization::make_nvp("foo",foo)`
- `BOOST_SERIALIZATION_NVP(foo)`

Hello phrase revisited

```
1  #include <iostream>
2  #include <string>
3  struct hello_phrase
4  {
5    explicit hello_phrase(std::string const &
6                          whom="World"): m_whom(whom){}
7    void say_it(std::ostream & s){ /* ... */ }
8    template<typename Archive>
9    void serialize(Archive & ar, const unsigned int)
10   {
11     ar & m_whom;
12   }
13   std::string m_whom;
14 };
```

Hello phrase revisited

```
1  #include <iostream>
2  #include <string>
3  #include <boost/serialization/nvp.hpp>
4  struct hello_phrase
5  {
6    explicit hello_phrase(std::string const &
7                          whom="World"): m_whom(whom){}
8    void say_it(std::ostream & s){ /* ... */ }
9    template<typename Archive>
10   void serialize(Archive & ar, const unsigned int)
11   {
12     ar & BOOST_SERIALIZATION_NVP(m_whom);
13   }
14   std::string m_whom;
15 };
```

XML Output

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive"
                     version="4">
<hp class_id="0" tracking_level="0" version="0">
        <m_whom>Sohail</m_whom>
</hp>
</boost_serialization>
```

How does that work?

- `ar << foo` calls global save function

```
template<typename Archive, typename T>
void save(Archive & ar, T const & t);
```

- Global save function calls global serialize function

```
template<typename Archive, typename T>
void serialize(Archive & ar, T const & t,
               const unsigned int fv);
```

- Default global serialize function calls member function serialize

```
template<typename Archive>
void T::serialize(Archive & ar,
                  const unsigned int fv);
```

- Similarly for `ar >> foo`
- Note: Do **not** depend on this behaviour!

Non-intrusive serialization

Recall that a type T is Serializable if it has a global function named serialize:

```
1  // Put in this namespace for maximum portability
2  namespace boost { namespace serialization {
3  template<typename Archive>
4  void serialize(Archive & ar, hello_phrase & hp,
5                 const unsigned int)
6  {
7    ar & BOOST_SERIALIZATION_NVP(hp.m_whom);
8  }
9  }}
```

Not exactly non-intrusive...

Non-default constructors

- Previous examples deserialized into fully-constructed instances
- Possible if there is a default constructor
- If no default constructor $\longrightarrow$ need to serialize and deserialize by pointer

Bank account

```
1  struct bank_account
2  {
3    // Note: No default constructor
4    bank_account(size_t id, double initial_balance);
5
6    size_t get_id() const;
7    double get_balance() const;
8    timestamp get_creation_time() const;
9    // other functions
10 };
```

Load/save construction data

```
1  // important for portability
2  namespace boost { namespace serialization {
3  template<typename Archive>
4  void save_construct_data(Archive & ar,
5                           bank_account const * b,
6                           const unsigned int);
7
8  template<typename Archive>
9  void load_construct_data(Archive & ar,
10                          bank_account * b,
11                          const unsigned int);
12
13 template<typename Archive>
14 void bank_account::serialize(Archive & ar,
15                               const unsigned int);
16 }}
```

save_construct_data

Save all the information needed to reconstruct

```
1  template<typename Archive>
2  void save_construct_data(Archive & ar,
3                           bank_account const * b,
4                           const unsigned int)
5  {
6    int const id = b->get_id();
7    double const balance = b->get_balance();
8    ar
9      & make_nvp("id",id)
10     & make_nvp("balance",balance);
11 }
```

load_construct_data

```
1  template<typename Archive>
2  void load_construct_data(Archive & ar,
3                           bank_account * b,
4                           const unsigned int)
5  {
6    size_t id;
7    double balance;
8
9    ar
10     & make_nvp("id",id)
11     & make_nvp("balance",balance);
12
13   ::new(b) bank_account(id,balance);
14 }
```

serialize

```
1  // Write serialize as usual
2  template<typename Archive>
3  void bank_account::serialize(Archive & ar,
4                               const unsigned int fv)
5  {
6    using boost::serialization::make_nvp;
7    ar & make_nvp("id",m_id)
8       & make_nvp("balance",m_balance)
9       & make_nvp("creation",m_creation_time);
10 }
```

Serializing an instance

```
1  #include "archive.hpp"
2  #include <sstream>
3
4  // bank_account defined here
5  int main()
6  {
7    using boost::serialization::make_nvp;
8    bank_account b1(10,5e6);
9    bank_account * pb1=&b1;
10   std::ostringstream os;
11   {
12     oarchive_t oa(os);
13     oa & make_nvp("bank_account",pb1);
14   }
15   std::cout << os.str() << std::endl;
16 }
17 // Output: 22 serialization::archive 4 0 1 0
18 // 0 10 5000000
```

De-serializing an instance

```
1  #include "archive.hpp"
2  #include <sstream>
3
4  // bank_account defined here
5  int main()
6  {
7    using boost::serialization::make_nvp;
8    // last slide's code goes here
9    bank_account * pb2 = 0;
10   {
11     std::istringstream is(os.str());
12     iarchive_t ia(is);
13     ia & make_nvp("bank_account",pb2);
14   }
15   delete pb2; // REMEMBER TO DELETE!
16 }
```

Streamable types

- Third-party types may sometimes be streamed to/from a C++ stream
- If type does not have a default constructor, use `load_construct_data` and `save_construct_data`

save_construct_data

```
1  template<typename Archive>
2  void save_construct_data(Archive & ar,
3                           TPType const * t,
4                           unsigned int fv)
5  {
6    std::ostringstream stream;
7    stream << t;
8    std::string contents(stream.str());
9    ar & make_nvp("streamed",contents);
10 }
```

load_construct_data

```
1  template<typename Archive>
2  void load_construct_data(Archive & ar,
3                           TPType * t,
4                           unsigned int fv)
5  {
6    std::string contents;
7    ar & make_nvp("streamed",contents);
8    std::istringstream is(contents);
9    ::new(t) TPType(is);
10 }
```

Streamable types

- If streamable type has a default constructor, then "split" serialize function

Splitting free serialize function

```
1  #include "boost/serialization/split_free.hpp"
2
3  // Must be outside any namespace
4  BOOST_SERIALIZATION_SPLIT_FREE(TPType);
5
6  namespace boost { namespace serialization {
7
8  template<typename Archive>
9  void save(Archive & ar, TPType & t,
10           const unsigned int)
11 {
12   std::ostringstream os;
13   os << t;
14   std::string contents(os.str());
15   ar & make_nvp("streamed",contents):
16 }
17
18 }}
```

Splitting free serialize function

```
1  namespace boost { namespace serialization {
2
3  template<typename Archive>
4  void load(Archive & ar, TPType & t,
5            const unsigned int)
6  {
7    std::string contents;
8    ar & make_nvp("streamed",contents);
9    std::istringstream is(contents);
10   is >> t;
11 }
12
13 }}
```

Splitting serialize member function

```
1  #include "boost/serialization/split_member.hpp"
2  ...
3  class foo
4  {
5     ...
6     BOOST_SERIALIZATION_SPLIT_MEMBER();
7     template<typename Archive>
8     void load(Archive & ar, const unsigned int){...}
9
10    template<typename Archive>
11    void save(Archive & ar, const unsigned int){...}
12    ...
13 };
```

BOOST_SERIALIZATION_SPLIT_FREE

```
1  BOOST_SERIALIZATION_SPLIT_FREE(T)
2
3  // equivalent to:
4  template<typename Archive>
5  void serialize(Archive & ar, T & t,
6                 const unsigned int fv)
7  {
8    boost::serialization::split_free(ar,t,fv);
9  }
10
11 // equivalent to:
12 template<typename Archive>
13 void serialize(Archive & ar, T & t,
14                const unsigned int fv)
15 {
16   if(Archive::is_saving::value) save(ar,t,fv);
17   else if(Archive::is_loading::value) load(ar,t,fv);
18 }
```

BOOST_SERIALIZATION_SPLIT_MEMBER

```
1  BOOST_SERIALIZATION_SPLIT_MEMBER()
2
3  // equivalent to:
4  template<typename Archive>
5  void serialize(Archive & ar, const unsigned int fv)
6  {
7    boost::serialization::split_member(ar,*this,fv);
8  }
9
10 // equivalent to:
11 template<typename Archive>
12 void serialize(Archive & ar, const unsigned int fv)
13 {
14   if(Archive::is_saving::value)
15     this->save(ar,fv);
16   else if (Archive::is_loading::value)
17     this->load(ar,fv);
18 }
```

Project time!

Project time!

Download

- Boost 1.35: `http://www.boost.org/users/download`
- Extract somewhere (and remember where you extracted it!) Call this place **boost-root**

Patches

- `http://taggedtype.net/~sohail/boost/patches`
- Download above directory to **boost-root**/patches
- If you've got wget:

```
$ wget http://taggedtype.net/~sohail/boost/patches/ \
 --recursive --level=1 --accept=.patch --cut-dirs=3 \
 -nH -P patches
```

Patch 1 (for G++ 4.x)

- Serialization lib shipped with some minor bug for g++ 4.x
- There is a fix! Found it too late unfortunately
- Apply:

```
$ cd boost-root
$ patch -p1 < patches/serialization_1_35_0_adl.patch
```

- Of course, patch fails...
- Change boost/serialization/export.hpp, line 101:
- `instantiate_ptr_serialization((T*)0, 0, adl_tag());`

Patch 2 (for sanity)

- Jamfile
- Apply:

```
$ cd boost-root/libs/serialization/test
$ patch -p0 < \
../../../patches/serialization_1_35_0_test_Jamfile.patch
```

- Or...
- Add following at line 13 of `libs/serialization/test/Jamfile.v2`:
- `BOOST_ARCHIVE_LIST = [ modules.peek : BOOST_ARCHIVE_LIST ] ;`

Build

```
$ cd boost-root/tools/jam/src
$ ./build.sh   # or build.bat
$ cd bin.*
$ export PATH=$PATH:`pwd` # or
$ set PATH=%PATH%;%CD%
$ cd ../../../../ # (up 4 levels)
$ export BOOST_ROOT=`pwd` # or
$ set BOOST_ROOT=%CD%
$ bjam --with-serialization --layout=system \
  variant=debug link=shared stage
```

- Optionally, append `-jN+1` where N is number of processors
- Unix: `stage/lib/libboost_[w]serialization-mt-d.so`
- Windows: `stage/lib/boost_[w]serialization-mt-gd.(dll|lib)`
- For more information, see "Getting started:" `http://boost.org/doc/libs/1_35_0/more/getting_started/windows.html`

Run some tests

```
$ cd libs/serialization/test
$ bjam variant=debug \
-sBOOST_ARCHIVE_LIST="text_archive.hpp"
```

Congratulations!

You are now a Boost Jam expert!

Congratulations!

You are now a Boost Jam expert!
Only kidding...

Project stubs

- http://taggedtype.net/~sohail/boostcon_projects.zip
- http://taggedtype.net/~sohail/boostcon_projects.tar.gz

```
$ cd /path/to/unzipped/projects
$ bjam --layout=system
```

Serializing a container

- Write serialization for `std::vector<T>`
- Test above with `std::vector<int>`
- Bonus points: Create serialization for a class of your own with no default constructor and serialize `std::vector<YourClass>`. Does it work?
- Things to consider:
  - Non-intrusive
  - Types with and without default constructors
  - No dynamic memory allocation
  - Optimizations?

Serializing multiple pointers to the same object

```
1  int main()
2  {
3    using boost::serialization::make_nvp;
4    bank_account b1(10,5e6);
5    bank_account * pb1=&b1;
6    std::ostringstream os;
7    {
8      oarchive_t oa(os);
9      oa & make_nvp("bank_account",pb1);
10     // Serialize same pointer twice
11     oa & make_nvp("bank_account",pb1);
12   }
13   std::cout << os.str() << std::endl;
14 }
15 // Output: 22 serialization::archive 4 0 1 0
16 // 0 10 5000000 0
17 // 0
```

De-serializing multiple pointers to the same object

```
1  int main()
2  {
3    // Last slide's code...
4    bank_account
5      *pb2 = 0,
6      *pb3 = 0;
7    {
8      std::istringstream is(os.str());
9      iarchive_t ia(is);
10     ia & make_nvp("bank_account",pb2);
11     ia & make_nvp("bank_account",pb3);
12   }
13
14   std::cout << std::boolalpha << (pb2!=0 && (pb2==pb3))
15             << std::endl;
16
17   delete pb2; // or delete pb3!
18 }
19 // Output: true
```

A little more information

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive"
                     version="4">
<bank_account class_id="0" tracking_level="1" version="0"
              object_id="_0">
        <id>10</id>
        <balance>5000000</balance>
        <creation>55</creation>
</bank_account>
<bank_account class_id_reference="0"
              object_id_reference="_0"></bank_account>
</boost_serialization>
```

Serialization traits

- Alter the way data is serialized
- Traits:
  - Version
  - Implementation level
  - Object tracking
  - Abstract class
  - Is wrapper
  - type_info implementation

Disabling object tracking

Object tracking is a trait

```
1  BOOST_CLASS_TRACKING(my_class,
2                       boost::serialization::track_never)
```

boost::shared_ptr<T>

Among other things, includes support for Boost's smart pointers:

```
1  ...
2  #include <boost/serialization/shared_ptr.hpp>
3  ...
4  int main()
5  {
6    ...
7    typedef boost::shared_ptr<bank_account>
8      bank_account_ptr_t;
9
10   bank_account_ptr_t pb1(new bank_account(10,5e6));
11   // Serialization code as before
12   ...
13   bank_account_ptr_t pb2;
14   // Deserizliation code as before
15   // Remove delete pb2 statement
16   // Correct use count maintained
17 }
```

Serializing boost::shared_ptr<T>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive"
                     version="4">
<bank_account class_id="0" tracking_level="0" version="1">
        <px class_id="1" tracking_level="1"
                         version="0" object_id="_0">
                <id>10</id>
                <balance>5000000</balance>
        </px>
</bank_account>
</boost_serialization>
```

Polymorphic classes

- Polymorphic class: Contains atleast one virtual function
- Type registration: required when serializing/deserializing by base pointers
- One exception to rule them all: `unregistered_class`

Registering base/derived relation

- System needs to be able to cast between base and derived classes
- `boost::serialization::base_object<Base>(Derived&)`
- `boost::serialization::void_cast_register<Derived,Base>()`
- `BOOST_SERIALIZATION_BASE_OBJECT_NVP(Base)`

Base class

```
1  #include "boost/serialization/is_abstract.hpp"
2
3  struct shape
4  {
5    virtual ~shape(){}
6    virtual double area() const = 0;
7
8    template<typename Archive>
9    void serialize(Archive &,
10                  const unsigned int){}
11 };
12
13 BOOST_IS_ABSTRACT(shape);
```

Triangle

```
1  struct triangle : public shape
2  {
3    triangle(double b, double h);
4    virtual double area() const;
5
6  private:
7    // Simplest way is to implement default constructor.
8    // But don't make it part of the public interface
9    friend class boost::serialization::access;
10   triangle(){}
11   template<typename Archive>
12   void serialize(Archive & ar,
13                  const unsigned int)
14   {
15     ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(shape)
16        & BOOST_SERIALIZATION_NVP(m_b)
17        & BOOST_SERIALIZATION_NVP(m_h);
18   }
19 ...
20 };
```

Square

```
1  struct square : public shape
2  {
3    square(double w);
4    virtual double area() const;
5
6  private:
7    friend class boost::serialization::access;
8    square(){}
9    template<typename Archive>
10   void serialize(Archive & ar,
11                  const unsigned int)
12   {
13     ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(shape)
14        & BOOST_SERIALIZATION_NVP(m_w)
15   }
16 ...
17 };
```

Serializing polymorphic classes by concrete type

```
1  // Business as usual
2  int main()
3  {
4    using boost::serialization::make_nvp;
5
6    typedef boost::shared_ptr<square> square_ptr_t;
7    typedef boost::shared_ptr<triangle> triangle_ptr_t;
8
9    std::ostringstream os;
10   {
11     square_ptr_t s(new square(5));
12     triangle_ptr_t t(new triangle(5,3));
13
14     oarchive_t oa(os);
15     oa & make_nvp("square",s);
16     oa & make_nvp("triangle",t);
17   }
18   ...
19 }
```

Deserializing polymorphic classes by concrete type

```
1  int main()
2  {
3    // last slide's code here
4    {
5      square_ptr_t s;
6      triangle_ptr_t t;
7
8      std::istringstream is(os.str());
9      iarchive_t ia(is);
10     ia & make_nvp("shape",s);
11     ia & make_nvp("shape",t);
12   }
13 }
```

Serializing polymorphic classes by base class pointer

```
1  int main()
2  {
3    using boost::serialization::make_nvp;
4
5    typedef boost::shared_ptr<shape> shape_ptr_t;
6
7    std::ostringstream os;
8    {
9      shape_ptr_t s1(new square(5)),
10       s2(new triangle(5,3));
11
12     oarchive_t oa(os);
13     oa & make_nvp("shape",s1);
14     oa & make_nvp("shape",s2);
15   }
16 ...
17 }
```

De-serializing polymorphic classes by base class pointer

```
1  int main()
2  {
3    // last slide's code
4    {
5      shape_ptr_t s1, s2;
6
7      std::istringstream is(os.str());
8      iarchive_t ia(is);
9      ia & make_nvp("shape",s1);
10     ia & make_nvp("shape",s2);
11   }
12 }
13 // Oops:
14 //   terminate called after throwing an instance of
15 //   'boost::archive::archive_exception'
16 //    what():  unregistered class
```

Class registration

- Up until now, we have been enjoying class-registration as a side-effect
- `ar & foo` $\longrightarrow$ If `typeof(foo)` is not registered with the archive, it is registered and given a sequential ID
- An important result: You must serialize and deserialize in the exact same order!
- When serializing by base class pointer, `typeof(foo)` is `base` and **not** `child`

Registering classes the hard way

Recall that `ar.register_type<T>()` registers the type with the archive.

```
1  ...
2    std::ostringstream os;
3    {
4      shape_ptr_t s1(new square(5)),
5        s2(new triangle(5,3));
6
7      oarchive_t oa(os);
8
9      oa.register_type<square>();    // cid = 0
10     oa.register_type<triangle>(); // cid = 1
11
12     oa & make_nvp("shape",s1);     // cid = 2,3
13     oa & make_nvp("shape",s2);     // cid = 2,3
14   }
15 ...
```

Registering classes the hard way

Similarly for deserialization:

```
1  ...
2    {
3      shape_ptr_t s1, s2;
4
5      std::istringstream is(os.str());
6      iarchive_t ia(is);
7
8      ia.register_type<square>();   // cid = 0
9      ia.register_type<triangle>();// cid = 1
10
11     ia & make_nvp("shape",s1);    // cid = 2,3
12     ia & make_nvp("shape",s2);    // cid = 2,3
13
14     std::cout << typeid(*s1).name() << " "
15               << typeid(*s2).name() << std::endl;
16   }
17 ...
```

Class IDs from previous example

```
...
<shape class_id="2" tracking_level="0" version="1">
        <px class_id="0" tracking_level="1"
                         version="0" object_id="_0">
                <shape class_id="3" tracking_level="0"
                                    version="0"></shape>
                <m_l>5</m_l></px></shape>
<shape>
        <px class_id="1" tracking_level="1"
                         version="0" object_id="_1">
                <shape></shape>
                <m_w>5</m_w>
                <m_h>3</m_h></px></shape>
...
```

Exporting classes

- Manually registering classes is not ideal
- Another mechanism provided: `BOOST_CLASS_EXPORT(T)`
- Associates a string literal with a type (and corresponding deserializer)
- Problem: system relies on templates with arguments `Archive,T`. Example:

```
1  template<typename Archive, typename T>
2  void serialize(Archive &, T &, const unsigned int);
```

- Need to instantiate all combinations of `Archive` and `T`
- `BOOST_CLASS_EXPORT(T)` handles this
- Equivalent to: `BOOST_CLASS_EXPORT_GUID(T,"T")`

The new and improved archive.hpp

```
1  #ifndef INCLUDED_ARCHIVE_HPP
2  #define INCLUDED_ARCHIVE_HPP
3
4  #include "boost/archive/xml_oarchive.hpp"
5  #include "boost/archive/xml_iarchive.hpp"
6  #include "boost/serialization/export.hpp" // new
7
8  // Too lazy to include these everywhere ;-)
9  #include "boost/serialization/nvp.hpp"
10 #include "boost/serialization/base_object.hpp"
11
12 typedef boost::archive::xml_oarchive oarchive_t;
13 typedef boost::archive::xml_iarchive iarchive_t;
14
15 #endif
```

shape.hpp

```
1  struct shape
2  {
3    virtual ~shape(){}
4    virtual double area() const = 0;
5
6    template<typename Archive>
7    void serialize(Archive &,
8                   const unsigned int){}
9  };
```

triangle.hpp

```
1  #include "shape.hpp"
2  #include "boost/serialization/access.hpp"
3
4  struct triangle : public shape
5  {
6    triangle(double w, double h);
7    virtual double area() const;
8
9  private:
10   double m_w, m_h;
11
12   triangle(){}
13   friend class boost::serialization::access;
14   template<typename Archive>
15   void serialize(Archive & ar,const unsigned int);
16 };
```

triangle.cpp

```
1  #include "triangle.hpp"
2  #include "archive.hpp"
3
4  ... other member functions ...
5  template<typename Archive>
6  void triangle::serialize(Archive & ar,
7                           const unsigned int)
8  {
9    ar
10     & BOOST_SERIALIZATION_BASE_OBJECT_NVP(shape)
11     & BOOST_SERIALIZATION_NVP(m_w)
12     & BOOST_SERIALIZATION_NVP(m_h);
13 }
14
15 // Instantiates serialization for
16 // Archive=oarchive_t, T=triangle
17 // Archive=iarchive_t, T=triangle
18 BOOST_CLASS_EXPORT(triangle)
```

Serializing polymorphic classes the easy way

```
1  int main()
2  {
3    using boost::serialization::make_nvp;
4
5    typedef boost::shared_ptr<shape> shape_ptr_t;
6
7    std::ostringstream os;
8    {
9      shape_ptr_t s1(new square(5)),
10       s2(new triangle(5,3));
11
12     oarchive_t oa(os);
13
14     oa & make_nvp("shape",s1);
15     oa & make_nvp("shape",s2);
16   }
17   ...
18 }
```

Deserializing polymorphic classes the easy way

```
1  int main()
2  {
3    // last slide's code here...
4    {
5      shape_ptr_t s1, s2;
6
7      std::istringstream is(os.str());
8      iarchive_t ia(is);
9
10     ia & make_nvp("shape",s1);
11     ia & make_nvp("shape",s2);
12
13     std::cout << typeid(*s1).name() << " "
14               << typeid(*s2).name() << std::endl;
15   }
16 }
```

Serializing class templates

- Nothing special needs to occur
- Unless they are serialized by base class pointer

composite.hpp

```
1  template<typename S1, typename S2>
2  struct composite : public shape
3  {
4    composite(S1 s1, S2 s2){...}
5    virtual double area() const{...}
6  private:
7    friend class boost::serialization::access;
8    composite(){}
9    template<typename Archive>
10   void serialize(Archive & ar,const unsigned int)
11   {
12     ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(shape)
13       & BOOST_SERIALIZATION_NVP(m_s1)
14       & BOOST_SERIALIZATION_NVP(m_s2);
15   }
16   boost::shared_ptr<S1> m_s1;
17   boost::shared_ptr<S2> m_s2;
18 };
```

Serializing composite

- Problem: `BOOST_CLASS_EXPORT(T)` cannot work for class templates
- Solution: Need to export each instantiation of `composite<S1,S2>`

Serializing composite

```
1  ...
2  #include "composite.hpp"
3  ...
4  typedef composite<square,triangle> comp_sq_tr_t;
5
6  BOOST_CLASS_EXPORT(comp_sq_tr_t);
7  ...
8  int main()
9  {
10   ...
11   {
12     shape_ptr_t s1(new comp_sq_tr_t(square(5),
13                                     triangle(5,3)));
14     oarchive oa(os);
15     oa & make_nvp("shape",s1);
16   }
17   ...
18 }
```

Project time!

Project time!

Serializing polymorphic function objects

- We want to write a RPC framework
- For it to work, the receiving end cannot know the concrete type of the job to be executed.
- Specifically, the RPC server can only send back a string (which can handily be the serialized result)
- Things to try:
  - Try doing this one with manual registration as well as automatic registration as it may help you on your next project!
  - Exceptions

Serializing standard C++ objects

Almost too easy:

```
1  #include "boost/serialization/vector.hpp"
2  ...
3  int main()
4  {
5    using boost::serialization::make_nvp;
6    std::ostringstream os;
7    {
8
9      std::vector<shape_ptr_t> l;
10     for(size_t i =0 ; i < 10; ++i)
11       l.push_back(shape_ptr_t(new triangle(double(i+1),3)));
12
13     oarchive_t oa(os);
14     oa & make_nvp("vector",l);
15   }
16   ...
17 }
```

Supported external classes

- `complex`
- `pair`
- All C++ standard containers and some extensions (`hash_*`)
- `boost::optional`
- `boost::shared_ptr`
- `boost::weak_ptr`
- `boost::scoped_ptr`
- `boost::variant`
- Various other Boost libraries natively provide support:
  `boost::multi_index`, `boost::date_time`

Choosing the right Archive

- Decide on your criteria:
  - Speed of serialization: binary
  - Archive output size: binary, text
  - Archive output readability: text, xml
  - Archive portability: text, xml. Note: **not binary!**
  - Code-bloat: polymorphic
  - Swapping out at runtime: polymorphic

archive.hpp

```
1  // Need to avoid instantiating unncessary serialization code
2  // so don't include concrete archives
3  // If we include concrete archives, that causes
4  // unnecessary code bloat
5  #include "boost/archive/polymorphic_oarchive.hpp"
6  #include "boost/archive/polymorphic_iarchive.hpp"
7
8  #include "boost/serialization/export.hpp"
9  #include "boost/serialization/nvp.hpp"
10 #include "boost/serialization/base_object.hpp"
```

shape.hpp

```
1  // No need to change this
2  struct shape
3  {
4     virtual ~shape(){}
5     virtual double area() const = 0;
6
7     template<typename Archive>
8     void serialize(Archive &,
9                    const unsigned int){}
10 };
```

triangle.hpp

```
1  // No real need to change this either!
2  #include "shape.hpp"
3  #include "boost/serialization/access.hpp"
4
5  struct triangle : public shape
6  {
7     triangle(double w, double h);
8     virtual double area() const;
9
10 private:
11    double m_w, m_h;
12
13    triangle(){}
14    friend class boost::serialization::access;
15    template<typename Archive>
16    void serialize(Archive & ar,const unsigned int);
17 };
```

triangle.cpp

```
1  // See a pattern? No need to change this either!
2  #include "triangle.hpp"
3  #include "archive.hpp"
4
5  ... other member functions ...
6  template<typename Archive>
7  void triangle::serialize(Archive & ar,
8                           const unsigned int)
9  {
10   ar
11     & BOOST_SERIALIZATION_BASE_OBJECT_NVP(shape)
12     & BOOST_SERIALIZATION_NVP(m_w)
13     & BOOST_SERIALIZATION_NVP(m_h);
14 }
15 // Instantiates serialization for
16 // Archive=polymorphic_oarchive_t, T=triangle
17 // Archive=polymorphic_iarchive_t, T=triangle
18 BOOST_CLASS_EXPORT(triangle)
```

concrete_polymorphic_archives.hpp

```
1  #include "boost/archive/polymorphic_text_oarchive.hpp"
2  #include "boost/archive/polymorphic_text_iarchive.hpp"
3
4  #include "boost/archive/polymorphic_binary_oarchive.hpp"
5  #include "boost/archive/polymorphic_binary_iarchive.hpp"
```

Choosing output archive at runtime

```
1  #include "concrete_polymorphic_archives.hpp"
2  ...
3    bool use_text_archive = get_bool_from_user();
4    std::ostringstream os;
5    {
6      namespace ba = boost::archive;
7      shape_ptr_t s1(new triangle(5,3));
8
9      std::auto_ptr<ba::polymorphic_oarchive> poa;
10     if(use_text_archive)
11       poa.reset(new ba::polymorphic_text_oarchive(os));
12     else
13       poa.reset(new ba::polymorphic_binary_oarchive(os));
14
15     *poa & make_nvp("shape",s1);
16   }
17 ...
```

Choosing input archive at runtime

```
1  #include "concrete_polymorphic_archives.hpp"
2  ...
3    {
4      namespace ba = boost::archive;
5      std::istringstream is(os.str())
6      shape_ptr_t s1;
7
8      std::auto_ptr<ba::polymorphic_iarchive> pia;
9      if(use_text_archive)
10       pia.reset(new ba::polymorphic_text_iarchive(is));
11     else
12       pia.reset(new ba::polymorphic_binary_iarchive(is));
13
14     *pia & make_nvp("shape",s1);
15   }
16 ...
```

Class Versioning

- Built-in backward compatibility
- No forward compatibility

Class Versioning

- To mark a class as being a new version for the purposes of serialization:

```
1  #include "boost/serialization/version.hpp"
2  class shape {...};
3  BOOST_CLASS_VERSION(shape, 2)
```

- To load in earlier versions, use version argument:

```
1  template<typename Archive>
2  void serialize(Archive & ar,
3                 const unsigned int version)
4  {
5    // versions greater than 2 have a name
6    if(version >= 2)
7      ar & m_name;
8    // Otherwise give a sensible default
9    // Note that this is a compile-time constant
10   else if (Archive::is_loading::value)
11     m_name = "Unnamed shape";
12 }
```

Wrappers

- Convenient to wrap data for specific type of serialization
- Example: pointer as array, pointer as binary data, treating an existing type as something else
- `make_binary_object(void * address,size_t size)`
- `make_array(T * t, size_t count)`
- `make_nvp(char * name, T & value)`
- `BOOST_STRONG_TYPEDEF(original,new)`

Binary wrapper

```
1  #include "boost/serialization/binary_object.hpp"
2  ...
3    namespace bs = boost::serialization;
4    T t;
5    // NOT PORTABLE!
6    ar & bs::make_binary_object(&t,sizeof(T));
7  ...
```

Arrays

Supports serialization of a contiguous sequence of a data type.

```
1  #include "boost/serialization/array.hpp"
2  ...
3    T t[10];
4    ar & bs::make_array(t,sizeof(t)/sizeof(T));
5  ...
6    boost::array<T,10> a = {...};
7    ar & bs::make_array(a.c_array(),a.size());
```

BOOST_STRONG_TYPEDEF

Supports distinguishing types for the purposes of serialization.

```
1  #include "boost/strong_typedef.hpp"
2
3  // An int thats always stored in big endian
4  BOOST_STRONG_TYPEDEF(int,big_endian_int);
5
6  template<typename Archive>
7  void load(Archive & ar, big_endian_int & b, const uint)
8  { ...
9    ar & static_cast<int &>(b); // avoid recursion
10 }
11
12 template<typename Archive>
13 void save(Archive & ar, big_endian_int const & b, const uint)
14 { ...
15   ar & static_cast<const int &>(b); // avoid recursion
16 }
17
18 BOOST_SERIALIZATION_SPLIT_FREE(big_endian_int);
19 ...
```

Project time!

Project time!

Secret strings!

- We have a typedef: `typedef std::string secret_string` that we need to encipher and decipher when serializing.
- Use simple cipher: add 1 to each character to encipher, minus 1 to decipher.

Using Archive output for application file format

Identify criteria:

- Portability
- Ease-of-use
- Backward-compatibility
- Forward-compatibility
- Possibility of exchange with other systems

Version 0: Using class versions to achieve backward compatibility

```
1  template<typename Archive>
2  void shape::serialize(Archive & ar,
3                        const unsigned int)
4  {}
```

Version 1: Split member

```
1  ...
2  // in shape.hpp
3  BOOST_CLASS_VERSION(shape,1)
4  ...
5  // in shape.cpp
6  template<typename Archive>
7  void shape::serialize(Archive & ar,
8                        const unsigned int fv)
9  {
10   boost::serialization::split_member(ar,*this,fv);
11 }
12 ...
```

Version 1: Save

```
1  template<typename Archive>
2  void shape::save(Archive & ar,
3                   const unsigned int fv) const
4  {
5    ar & make_nvp("name",m_name);
6  }
```

Version 1: Load

```
1  template<typename Archive>
2  void shape::load(Archive & ar,
3                   const unsigned int version)
4  {
5    if (version >= 1)
6      ar & make_nvp("name",m_name);
7    else
8      m_name = "Unnamed";
9  }
```

Version 2: Save

```
1  // In shape.hpp
2  BOOST_CLASS_VERSION(shape,2)
3  ...
4  // In shape.cpp
5  template<typename Archive>
6  void shape::save(Archive & ar,
7                   const unsigned int fv) const
8  {
9    ar & make_nvp("name",m_name);
10   ar & make_nvp("center",m_center);
11 }
```

Version 2: Load

```
1  template<typename Archive>
2  void shape::load(Archive & ar,
3                   const unsigned int version)
4  {
5    if (version >= 1)
6    {
7      ar & make_nvp("name",m_name);
8      if(version >= 2)
9        ar & make_nvp("center",m_center);
10     else
11       m_center = point(0,0);
12   }
13   else
14   {
15     m_name = "Unnamed";
16   }
17 }
```

Creating forward-compatible file formats

- Forward compatibility: Ability to accept input intended for later versions
- Ignoring unknown data
- Problem: Unsupported by Boost.Serialization

A Solution

- Serialization library consumes all input for a given archive (more or less)
- Let it consume all input, but we ignore some of it
- `std::map<std::string,std::string>`
- See `forward_compatible` example in projects package

Save helper function

```
1  template<typename T>
2  void save_helper(T const & source,
3                   std::string & target)
4  {
5    std::ostringstream os;
6    ba::text_oarchive nested(os,ba::no_header);
7    nested << source;
8    target = os.str();
9  }
```

Load helper function

```
1  template<typename T>
2  void load_helper(T & target,
3                   std::string const & source)
4  {
5    std::istringstream is(source);
6    ba::text_iarchive nested(is,ba::no_header);
7    nested >> target;
8  }
```

Version 2: Save

```
1  template<typename Archive>
2  void shape::save(Archive & ar,
3                   const unsigned int) const
4  {
5     std::map<std::string,std::string> out;
6     save_helper(m_center,out["center"]);
7     save_helper(m_name,out["name"]);
8     ar & make_nvp("map",out);
9  }
```

Version 1: Load

```
1  template<typename Archive>
2  void shape::save(Archive & ar,
3                   const unsigned int fv) const
4  {
5    std::map<std::string,std::string> in;
6    ar & make_nvp("map",in);
7    load_helper(m_name,in["name"]);
8  }
```

Output

81 bytes with header, 55 without.

```
22 serialization::archive 4 0 2 0 0 3 0 0 0 4
name 10 8 top-left 1 x 1 5 1 y 1 3
```

Archive flags

- Each archive has a second argument: flags
- Available flags (in `boost::archive` namespace):

```
1  enum archive_flags {
2    // suppress archive header info
3    no_header = 1,
4    // suppress alteration of codecvt facet
5    no_codecvt = 2,
6    // suppress checking of xml tags
7    no_xml_tag_checking = 4,
8    // suppress ALL tracking
9    no_tracking = 8
10 };
```

Another solution

... To be filled in later ...

Tips

- Deterimine requirements up front
  - Compilers
  - Thread-safety
  - Formats/cross-language communication
- Read the manual
- Read the source code

Compilers

- Serialization library is quite template heavy
- Run the serialization tests!
- Supported compilers: GCC, Intel, Visual C++, HP ACC
- Unsupported but probably work: SunCC
- Regression tests

Thread-safety

- Boost 1.35 and lower are not thread-safe
- Two archives registering an unregistered type in different threads → boom!
- Some workarounds available

Formats

- Archive output is proprietary to Boost Serialization!
- Not possible to talk to other non-boost parsers
- Doesn't need to be so...
- Criteria: portability, speed, space, readability

Project time!

Project time!

Lame-RPC

- The general syntax I want:

```
1  int add_ints(int a, int b);
2  ...
3  rpc::result<int> foo =
4    rpc::call<int>("my-server",
5                   boost::bind(add_ints,
6                               5,6));
7  cout << "Result is: " << foo.get_result() << endl;
```

- Things to consider:
  - How to erase the type of function being called but be able to serialize and deserialize it
  - "bind_serialize.hpp" serializes a boost bind expression (lightly-tested) so you don't have to!