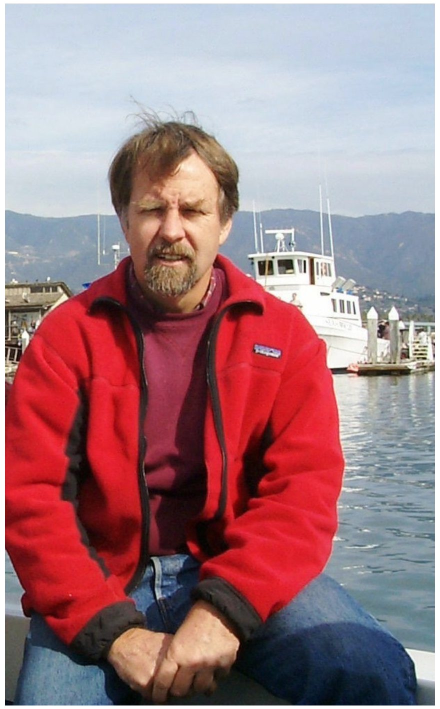
Boost.Serialization – a short introduction –

Developed by Robert Ramey
http://www.rrsd.com/



Serialization

- the reversible deconstruction of an arbitrary set of C++ data structures to a sequence of bytes

- Can be used for
  - Storing to and loading from a file
  - Remote parameter passing
  - Message passing
  - …

Archive

- a specific rendering of this stream of bytes.
- Examples
  - Binary file
  - Text file
  - XML file
  - Buffer in memory
  - MPI message
  - …

Requirements

- Non-intrusive
- Versioning for each class
- Deep pointer save and restore
- Proper restoration of pointers to shared data
- Enable data portability
- Orthogonal specification of class serialization and archive format

A first example

```
class Client
{
 public:
    Client()
    {}

    Client(int c, std::string r)
      : cif_number(c),
        rating(r)
    {}

 private:
    int cif_number;
    std::string rating;
};
```

```
std::ofstream ofs("filename");
boost::archive::text_oarchive  oa(ofs);


const Client c(123521,"AAA");
oa << c;
ofs.close();
```

```
std::ifstream ifs("filename");
boost::archive::text_iarchive  ia(ifs);

Client c;
ia >> c;
ifs.close();
```

Boost.Serialization overview

- Serialization of
  - Primitive types
  - Arrays
  - Pointers
  - Classes
  - Derived Classes through base class pointers
  - …

- Archive types
  - Text archive
  - Wide character text archive
  - Binary archive
  - XML archive
  - Wide character XML archive

- Additional archive types are not hard to implement

Boost.Serialization archives

- A (nearly) portable text archives
  - boost::archive::text_oarchive(ostream &s)
  - boost::archive::text_iarchive(istream &s)
  - boost::archive::text_woarchive(wostream &s)
    boost::archive::text_wiarchive(wistream &s)

- Non-portable native binary archives
  - boost::archive::binary_oarchive(ostream &s)
  - boost::archive::binary_iarchive(istream &s)

- XML archives
  - boost::archive::xml_oarchive(ostream &s)
  - boost::archive::xml_iarchive(istream &s)
  - boost::archive::xml_woarchive(wostream &s)
  - boost::archive::xml_wiarchive(wistream &s)

Intrusive Implementation

```
class Client
{
 public:
      Client();
      Client(int c, std::string r);

 private:
      int cif_number;
      std::string rating;

      friend class boost::serialization::access;

      template<class Archive>
      void serialize(Archive & ar, const unsigned int version)
      {
            ar & cif_number & rating;
      }
};
```

Less Intrusive Implementation

```
class Client
{
 public:
    Client();
    Client(int c, std::string r);

 public:
    int cif_number;
    std::string rating;

};
```

```
template<class Archive>
void serialize(
      Archive & ar,
      Client & g,
      const unsigned int version
    )
{
    ar & g.cif_number;
    ar & g.rating;
}
```

Derived classes

```
class PrivatePerson : public Client
{
 public:
    PrivatePerson();
    PrivatePerson(int c, std::string r, std::string n);

 private:
    std::string name;

    friend class boost::serialization::access;

    template<class Archive>
    void serialize(Archive & ar, const unsigned int version) {
          ar & boost::serialization::base_object< Client >(*this);
          ar & name;
    }
};
```

Serializing Pointers Works

```
boost::archive::text_oarchive  oa(std::ofstream("filename"));

const Client* cp1 = new Client (123521,”OK”);
const Client* cp2 = cp1;

oa << cp1;          // This serializes the object and pointer
oa << cp2;          // This serializes only the pointer
```

- When serializing objects
  - they are assigned an integer identifier
  - their address gets registered

- When serializing a pointer
  - The pointee is serialized if it has not been serialized yet
  - The identifier of the pointee is written to the archive

Deserializing Pointers Works

```
boost::archive::text_iarchive  ia(std::ifstream("filename"));

const Client* cp1 , cp2;

ia >> cp1;          // creates the object and deserializes the object and pointer
ia >> cp2;          // deserializes the pointer, pointing to the same object
```

- When deserializing objects
  - they are assigned an integer identifier
  - their address gets registered
- When deserializing a pointer
  - The identifier of the pointee is read from the archive
  - The pointee is deserialized if it has not been deserialized yet
  - The pointer is set

Serializing Arrays

```
class ClientArray
{
 public:
     ClientArray();

 private:
     Client * clients[1000];

     friend class boost::serialization::access;

     template<class Archive>
     void serialize(Archive & ar, const unsigned int version)
     {
          ar & clients;
     }
};
```

Serializing STL Collections

```
class ClientArray
{
 public:
      ClientArray();

 private:
      std::vector<Client*> clients;

      friend class boost::serialization::access;

      template<class Archive>
      void serialize(Archive & ar, const unsigned int version)
      {
            ar & clients;
      }
};
```

Calculated members

```
class FixedRate
{
 public:
     FixedRate(double a, double r);

 private:
     double amount, rate, interest;

     friend class boost::serialization::access;
     template<class Archive>
     void serialize(Archive & ar, const unsigned int)
     {
             ar & amount & rate & interest; // wastes space in archive
     }
};
```

```
FixedRate::FixedRate(double a, double r)
        : amount(a),
          rate(r),
          interest(a*r)
      {}
```

Calculated members

```
class FixedRate
{
 public:
     FixedRate(double a=0., double r=0.);

 private:
     double amount, rate, interest;

     friend class boost::serialization::access;
     template<class Archive>
     void serialize(Archive & ar, const unsigned int)
     {
           ar & amount & rate; // fails to recalculate interest on loading
     }
};
```

Splitting serialize

```
class FixedRate
{
 public:
      …
 private:
      …
      template<class Archive>
      void save(Archive & ar, const unsigned int) const
      {
            ar & amount & rate;                    // serialize amount and rate
      }

      template<class Archive>
      void load(Archive & ar, const unsigned int)
      {
            ar & amount & rate;                    // deserialize amount and rate
            interest = amount*rate;                // calculate interest
      }

      BOOST_SERIALIZATION_SPLIT_MEMBER()
};
```

Pointers to Objects of Derived Classes

```
class base
{
   ...
};

class derived_one : public base
{
   ...
};

class derived_two : public base
{
   ...
};
```

```
int main()
{
   ...
   base *b = new derived_one();
   oa  << b; // which derived class?
}
```

```
int main()
{
   ...
   base *b;
   ia  >> b; // which derived class?
}
```

Registering derived types

- Need to register all derived types

```
#include <boost/serialization/export.hpp>

BOOST_CLASS_EXPORT(derived_one)
BOOST_CLASS_EXPORT(derived_two)
```

- Note
  - Serialization of derived classes **must** call base class serialization even if that is empty

XML Serialization

```
class Client
{
 public:
      Client(
      Client(int c, std::string r);

 private:
      int number;
      std::string rating;

      friend class boost::serialization::access;
      template<class Archive>
      void serialize(Archive & ar, const unsigned version)
      {
            ar & BOOST_SERIALIZATION_NVP(number);
            ar & BOOST_SERIALIZATION_NVP(rating);
      }
};
```

$\longrightarrow$

```
<item class_id="0" ...>
    <number>132542</number>
    <rating>bankrupt</rating>
</item>
```

XML Serialization

```
class PrivatePerson : public Client
{
 public:
     PrivatePerson();
     PrivatePerson(int, std::string, std::string);

 private:
     std::string name;

     friend class boost::serialization::access;
     template<class Archive>
     void serialize(Archive & ar, const unsigned version)
     {
          ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP< Client >(*this);
          ar & BOOST_SERIALIZATION_NVP(name);
     }
};
```

$\longrightarrow$

```
<item class id="0" ...>
     <Client class_id="1" ...>
          <number>132542<number>
          <rating>bankrupt</rating>
     </Client>
     <name>Dave Abrahams</name>
</item>
```

Serialization traits: Implementation level

- Specified by specializing the **level** template, or macro call:

  BOOST_CLASS_IMPLEMENTATION(my_class, *value*)

- Possible values
  - boost::serialization::**not_serializable**
    -> do not serialize
  - boost::serialization::**primitive_type**
    -> archive knows how to serialize the type
  - boost::serialization::**object_serializable**
    -> call the object’s serialize function
  - boost::serialization::**object_class_info**
    -> store class info (version, name) and call the object’s serialize function

Object tracking

- Determines whether object addresses should be registered to allow serialization of pointers
- Specified by specializing the **tracking** template, or macro call:

  BOOST_CLASS_TRACKING(my_class, *value*)

- Possible values
  - boost::serialization::**track_never**
    -> never track the object’s address
  - boost::serialization::**track_selectively**
    -> track only if the type was explicitly registered
  - boost::serialization::**track_always**
    -> track always

New Versions of Classes

```
class Client
{
      ...
 private:
     int cif_number;
     std::string rating;

     friend class boost::serialization::access;

     template<class Archive>
     void serialize(Archive & ar, const unsigned int version)
     {
              ar & number & rating;
     }
};
```

New Versions of Classes

```
class Client
{
      ...
 private:
     int cif_number;
     std::string rating;
     bool simulated_client;
     friend class boost::serialization::access;

     template<class Archive>
     void serialize(Archive & ar, const unsigned int version)
     {
          ar & number & rating & simulated_client;
     }
};
```

- We can no longer read the old files!

Class Versioning

```
class Client
{
      ...
 private:
     int cif_number;
     std::string rating;
     bool simulated_client;
     friend class boost::serialization::access;

     template<class Archive>
     void serialize(Archive & ar, const unsigned int version)
     {
           ar & number & rating;
           if (version>0)
             ar & simulated_client
     }
};

BOOST_CLASS_VERSION(Client, 1)
```

Serialization traits: version

- Specified by specializing the **version** template

```
namespace boost { namespace serialization {

struct version<my_class>
{
    static const unsigned int value = 2;
};

}}
```

- Convenience macro

```
    BOOST_CLASS_VERSION(my_class, 2)
```

Classes without default constructor

```
Class Counterparty {
 public:
    Counterparty(int id)
      : identifier(id)
    {}
 private:
    int identifier;
}
```

```
Counterparty *c;
ia >> c;                        // fails!

// Boost.serialization  attempts:
c = new Counterparty();  // fails!
ia >> *c;

// Boost.serialization  should do:
int id;
ia >> id;
c = new Counterparty(id);
```

Overload construction of object

```
namespace boost { namespace serialization  {

template<class  Archive>
inline void load_construct_data
     (
            Archive & ar,
            Counterparty  * c,
            const unsigned version
     )
{
    int m;
    ar >> m; // retrieve  data from archive required to construct new instance
     ::new(c) Counterparty(m); // invoke  inplace constructor to initialize
}

}}
```

Serializing reference members

- Similar problem as before since reference members usually require non-default constructor

- Solution: serialize references as pointers and overload **save_construct_data** and **load_construct_data**

- Look at the documentation for details

Advanced topics

- See the documentation for information on
  - Creating your own archives
  - Serialization exceptions
  - Polymorphic archives
  - Portable archives
  - …