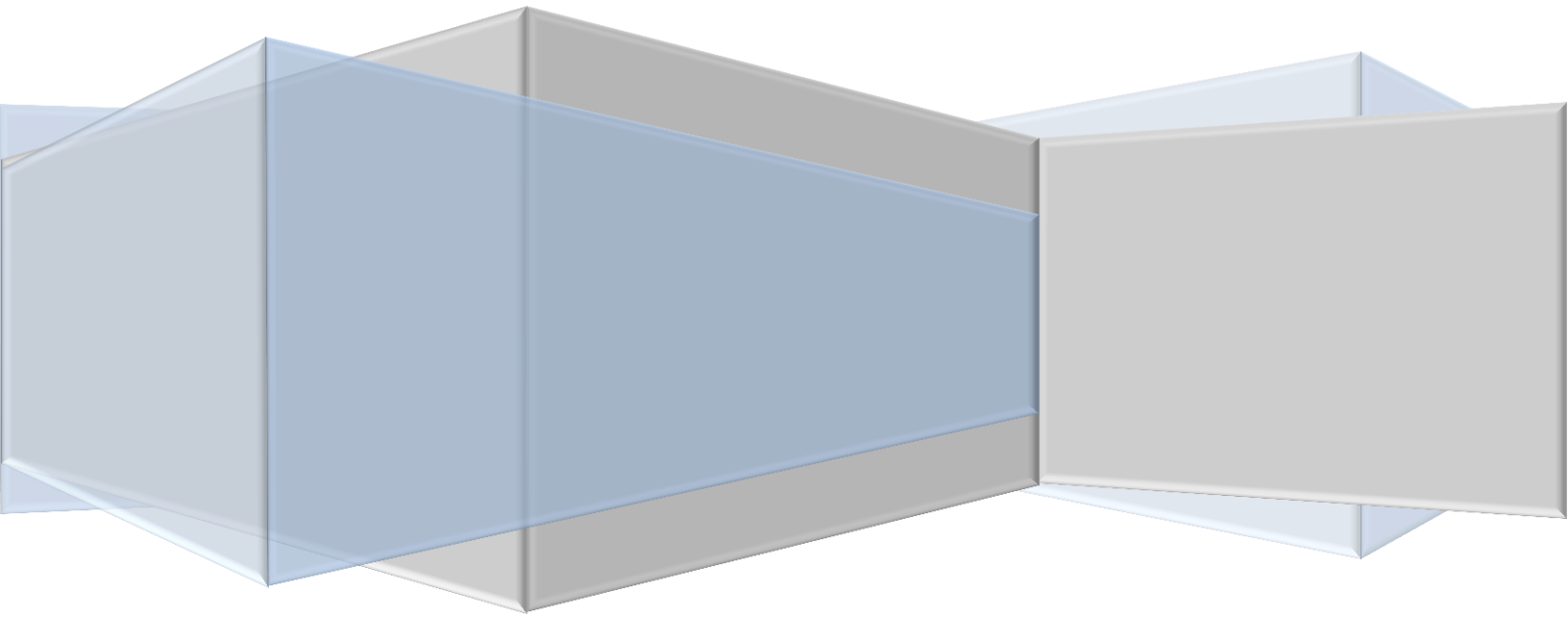
SERIALIZATION

Aparna Rajeev

Venkatesh Krishnamoorthy

Vijaya Senthil Veeri Chetty



Contents

Design Document.....

Introduction
Design
- Overview
- Self-Identifying streams
- STL Containers and Adaptors
- Containment
- Inheritance and virtual pointers:
- Pointers and Arrays
- Custom memory management:
- Traversal and Loops:

Implementation
Conclusion
Reference Manual.....
- Serializer class Reference
- Deserializer class Reference

Tutorial.....
- Features supported
- A simple serializable Class
- Containment
- Inheritance
- Pointers
- Arrays
- Legacy Classes

Reference.....

Design Document

Contents:

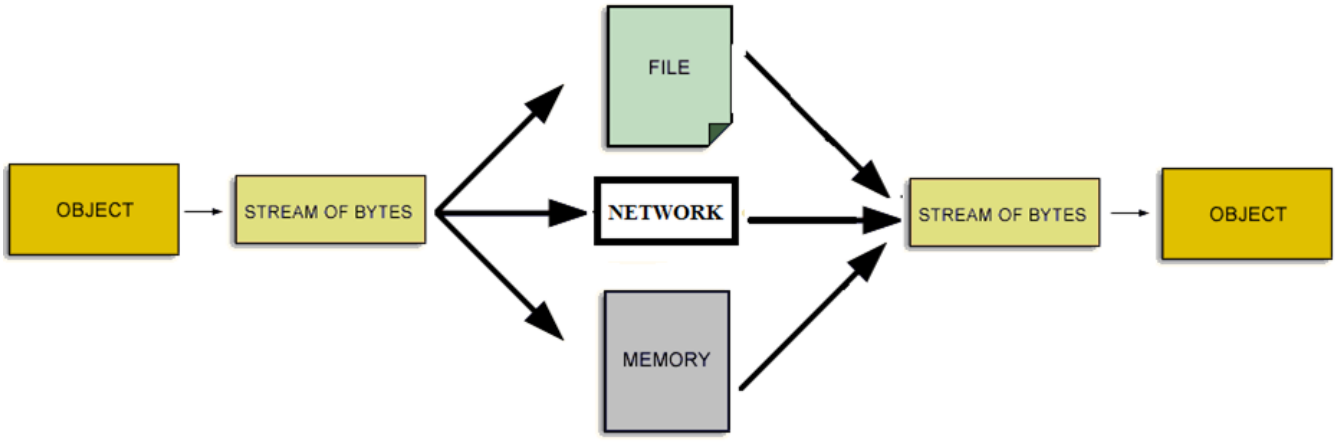
Introduction
Design
- Overview
- Self-Identifying streams
- STL Containers and Adaptors
- Containment
- Inheritance and virtual pointers:
- Pointers and Arrays
- Custom memory management:
- Traversal and Loops:

Implementation
Conclusion

Introduction:

Serialization is the process of converting a data structure or object into a format that can be “stored” (for example, in a file or memory buffer, or transmitted across network connection link) and "resurrected" later in the same or another computer environment. When the resulting series of bytes is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward [1].

Consider a server client architecture in which, both the parties have to send a set of information. This information would be stored inside objects of known type. When needed to send information, the sender has to extract individual entries it wants to send to client and either create a single stream containing data and push in a single transmission or send them individually using multiple transmissions. This will look obvious and simple if the amount of information that has to be stored is minimal say less than 10 fields per class. But when the class's structure grows or number of classes in the code repository grows, this becomes a tedious task. In this scenario, serialization comes to our help. It provides a set of minimal interfaces through which set of objects can be saved from one side and restored on the other side. Serialization is also helpful in lot of real life scenarios such as storing data structures locally and restoring when required, or sharing a set of objects in memory while jumping between multiple procedures without the need to write wrapper classes [2].



The system we have developed provides easy interfaces to serialize a set of objects and fetch the corresponding stream. This stream is a sequence of characters which when fed to deserializer interfaces restores a copy of original object.

Terminology:

Basic Type: Basic types include c data types such as int, float, char, double, long, bool, long double and short.

Primitive Type: Along with basic type support, it also contains support for all the standard container and adaptar types defined in Standard Template Library such as string, vector, list, map, multimap, deque, queue, standard array, priority queue, stack and forward_queue.

User Defined Type: A class or structure defined by the user containing both primitive typed members and user defined type members.

Push/saving objects: An instance of serializer is created. And an object/ptr is sent to the serializer using one of the save interfaces (save_object , save_array)

***Restore/Load/Resurrect Object*:** The stream created by an instance of serializer is used to initialize a deserializer. And user extracts the object from the stream using one of the Deserializer interfaces (load_object, load array).

Design - Overview:

The main issue with C++ objects serialization is that, it is impossible to retrieve an object's layout i.e. list of all the data members present in the class. So the user is expected to provide the list of member needed to be serialized for that class. And while deserializing, user extracts all the members that was pushed earlier.

Every class which wants it to be serialized implements two mandatory functions named ‘serialize’ and ‘deserialize’. The argument for serialize function would be the Serializer object into which it will push it members. And the argument for deserialize function is Deserialier object from which it can retrieve the members. To start of with, user creates a serializer object and uses “*save_object*” function of it to push the objects that needs to be serialized. In serializer, if the type of entity being pushed is primary type, we use specialized functions to access the values directly. If its type is one of the user defined type (an instance of say class A), the serialize member function of that class is called (call A's serialize function).

To deserialize a stream that was already generated by our serializer, user creates an instance of class Deserializer. It is mandatory to initialize Deserializer with a serializer generated stream. Deserializer's *load_object* interface takes the target object/entity as an argument and restores the serialized object into it. Similar to how serializer works, deserializer looks at the type of argument being passed. If it is a primitive type, value is directly loaded into the argument. If it is of User Defined Type, corresponding class's deserialize function is called.

Another obvious issue associated with this approach is that, whether the order of pushing the data members is important or is it oblivious to the order of retrieval. Here we go with the assumption that both the parties know the order of serialization and deserialization of data members for a particular class. For instance, for a class containing member 'a' of type int, 'b' of type string, if serializer pushes 'a' followed by 'b', deserializer should retrieve 'a' first and 'b' next. Changing the order while deserializing will result in undefined behavior.

Self-Identifying Streams:

When saving objects using Serializer interfaces (save_object, save_array), user has the option of providing a name/label to the object. These objects reach the receiver as self-identifying streams. If the receive is not sure about the object currently needs to be restored, receiver can choose to query the name of the object and use predefined mapping to create correct object instance before deserializing the stream. This will be helpful in the scenario where both the parties are aware of objects that will be transmitted, but not sure about the order in which objects are serialized.

STL containers and Adaptors:

Our serializer provides support for all of the STL containers and Adaptors. The list of datatypes supported by STL is fixed and clearly documented. So, when an argument of one of these types is passed, serializer and deserializer use precisely specialized functions to read/write data from/to these containers. For instance, if a map is passed, serializer gets the size of the map and stores it in the stream. Iterates through the map to get the pairs and stores the name and value in the stream. When deserializer tries to restore this map, it gets the size and fetches same number of name, value pairs from the stream and inserts into this new map. These name, value pairs would again belong to primary type or user defined type. Say it is a <uin, Student_Record> map. UIN is pushed using integer overloading and Student_Record is serialized by calling its serialize function. Deserialization happens in similar manner.

This pretty much covers the cases of serializing primitive types and simple user defined types. We will discuss the approaches involved as the complexity of user defined type increases.

Containment:

Both serialization and deserialization is handled recursively. This way multiple-levels of object containment is handled automatically. For instance,

```
Class Circle
{
      Point p;
      double radius;
```

```
        void serialize(Serilaizer& sr){
              sr << p <<radius;
        }
        void deserialize(Deserilaizer& dz){
              dz >> p >>radius;
        }
}
```

In this case, the user saves object of type Circle. Our Serializer invokes Circle::serialize. Here, user pushes just its member such as 'p' and 'radius'. When he pushes 'p', our Serializer identifies that 'p' belongs to another User Defined Type "Point" and calls serialize function of class Point. While deserializing similar approach is followed. Point class's deserialize is called to fill its members before filling member 'radius'.

Inheritance and virtual pointers:

Inheritance support comes with few restrictions. First of all, our library doesn't support serialization of classes having virtual pointer (base_ptr pointing to derived obj) as data members. If the class has a virtual pointer(base ptr) that points to a derived object, then serialization will not happen properly as the base object serialize function will be called. This may be resolved by making the serialize functions virtual. But at the deserializing end, the stream doesn't have information regarding which derived object was serialized. So resurrection might result in incorrect behavior.

Our library supports other types of inheritance.The user should not directly call the serialize and deserialize function of the base class as these functions can be private.The user needs to first push the base object by typecasting itself to a "base class" before pushing the data members of the derived class.

Pointers and Arrays:

So far we have discussed about instances of basic types, STL containers and user defined types. Our system supports serialization of pointers too. But, except char*, our library lacks support for storing pointers to basic types (int , float etc). So, saving a pointer will make the serializer fetch just a single instance of the data type. For eg: passing pointer to Student_Record just fetches one Student_Record object from memory and serializes it.

The lack of support for basic type pointers is under the assumption and reasoning that, there is no advantage in storing pointers to basic types unless it is an array. In that case, the array cannot operate just with single pointer, but will need a length value as well. *char** are different because, C-style char* are generally presumed to be null terminated. All other pointers to basic types should be the starting pointer to an array.

In order to support serialization of arrays of all data types (primitive and user defined type), serializer provides an interface called "*save_array(ptr, length)*". Serializer will iterate through the memory region and serialize all the individual elements of the array there by preventing data loss. For instance, *save_array (int* ptr, 10)* will fetch 10 integers from memory region starting from ptr and serializes. If it is "*save_array(Student_Record* ptr, 10)*, 10 student record objects are fetched and serialized.

Deserializing pointers is little bit tricky. Similar to save_object (ptr), load_object(ptr) just operates on one instance of the data type. This might result in data loss. So, another two interfaces named "get_array_size" and "load_array" are used in the same order to restore arrays. To restore an array, user queries the size of the array using "get_array_size" and allocates sufficient memory to restore the objects. Deserializer's "load_array" is used to restore the array, which loops through the length and restores same number of objects (basically, n times load_object).

Custom memory management:

Our serializer provides support for custom memory management. Whenever user tries to restore a pointer, our serializer doesn't use "new" by default. We understand that the use of default new might cause trouble in larger systems where custom memory management and garbage collection is employed. So in those scenarios, user has the option of defining a static function named "*allocate_memory*" for a particular class to do custom memory allocation. When user tries to restore pointer to an object of that class, deserializer calls this "*allocate_memory*" function and fills data in the memory returned by the function. If such a function is not provided in the class, we use default "new".

Traversal and Loops:

In most of the systems employing trees, graphs and linked lists, pointers play an important role in various operations. For instance, with just head/root pointer it is possible to traverse the entire data structure. As discussed above, our serializer uses recursion. For instance, consider the following tree_node structure.

```
Class tree_node
{
      int value;
      string color;
      tree_node *left;
      tree_node * right;
      void serialize(Serializer &s){
            s<<value<<color<<left<<right;
      }
      void deserialize (Deserializer &s){
            s>>value>>color>>left>>right;
    }
}
```

Most of the systems will just have the root pointer. So when this root pointer is sent to the serializer, it calls the serialize function in which say the data members are pushed. In the above example, value is pushed, then color , then left pointer and finally right pointer is pushed. When processing the left pointer, it is observed that it is pointer to a user defined type, so that node's serialize function is called ((*left).serialize). This will save the contents of left node and recursively continues this process. This way we do depth first traversal from the pointer given. Similar operation happens for Deserialization as well.

One obvious problem attached is redundant storage of pointers in case of joins and loops. Say pointers A&B points to the same memory location. In order to avoid this, serializer maintains a list of already seen pointers along with a unique ID. So if the pointer is encountered again, the

traversal is avoided and just the ID is stored in that place. While deserializing, whenever user tries to restore a pointer and we see an identifier stored in its place, we consult another map whether that ID was seen previously, if not new memory is allocated and <ID, pointer> pair is entered into the map before filling data in it. If that ID exists in the map, we simply copy the pointer and assign it to the requested pointer.

Implementation:

Like all serialization problems we started off with deciding a data structure to store the stream. The requirement is to add value and sufficient information about the arguments belonging to various type. And based on the type of argument used to retrieve, the additional information added is used to retrieve the values correctly. We needed a buffer to store the value and information. The "stringstream" type defined as part of the language was our choice. There are many advantages involved in using this data structure. First, it provides the exact functionalities we require from a buffer. It is well tested, and reduces lot of bugs during development. And finally,, as it is part of C++ standard, it is platform independent and avoids the cases where buffer operations differs on difference architectures (eg: endianness) and also, the optimizations developed (like compression etc) on this type will available to our system automatically. As mentioned in traversal and loops section, STL maps are used to keep track of pointers that were seen already. We have used "delete" feature of C++0X considerably to avoid certain overloads and specializations. And to give user the flexibility to add custom memory management, we have used meta programming to identify whether user's class has allocate_memory defined.

Using the serialized buffer:

Once all the serialization is done, user can ask the serializer to provide the buffer (stringstream) using "get_stream" interface. This way, the user can choose to perform variety of operations using the buffer ranging from saving to file, sending it over network, keeping it in memory etc.

On the other machine (or in the same machine when read from file), user initializes Deserializer using this buffer (stringstream). And restores a copy of original object using one or more deserializer interfaces.

Conclusion:

Our team developed a light-weight, platform independent serialization library which creates self-identifying stream and operates independent of platforms to convert almost any type of data and performs comparable to boost serialization library. Operations are discussed case by case and limitations and assumptions are specified appropriately.

Reference Manual

1. Serializer class Reference
2. Deserializer class Reference

Serialization Library:

Serialization library is defined by two complementary classes namely *Serializer* and *Deserializer*. Former class is for saving data while the latter is for loading it. To invoke serialization using our library, the user should include "Serialization.h" in the code module.

Serializer class Reference:

```
Class Serializer
{
protected:
        stringstream s;
        map<void*,int> ser_map;
public:
        template <class T>
         void  save_object(const T& t, const string name="dummy");
        template <class T>
         void  save_object(const T*& t, const string name="dummy");
        template <class T>
         void  save_array(const T* ptr , int n, const string name = "dummy");
        const stringstream& get_stream( );
};
```

Member Functions:

1. ***template <class T>***
   void save_object(const T& t, const string name="dummy");

   This function is for saving any primitive type or user defined type.

   Parameters:

   t : Any primitive type or user defined type as the first parameter.

   name : It takes name as the second optional parameter. The user can opt to pass the class name or any class identifier. This optional parameter helps the library to generate named streams. The user can ask for the class name before deserializing. This way the user need not remember the order in which the objects were serialized. The user can use this feature to have a mapping of class name to object construction.

2. ***template <class T>***
   void save_object(const T* & t, const string name="dummy");

   Same as above with first parameter being a pointer to user-defined type or char*. *Note*: The compiler will throw an error if you pass a pointer to any primitive type other than char *.

3. ***template <class T>***
   void save_array(const T* ptr , int n, const string name = "dummy");

   This function is for saving array of primitive types or user defined types.

   Parameters:

   ptr : It takes a pointer to the array of object/primitive type.

   n : Size of the array

name : The user can opt to pass the class name or any class identifier. This optional parameter helps the library to generate named streams.streams. The user can ask for the class name before deserializing. This way the user need not remember the order in which the objects were serialized. The user can use this feature to have a mapping of class name to object construction.

4. ***const stringstream& get_stream( );***

   The serializer serializes the object to a stringstream. The user can get this stringstream by calling get_stream. The user can opt to save the stringstream in a file or send it across the network.

Deserializer class Reference

```
Class Deserializer
{
       map<void*,int> deser_map;
public:
       Deserializer (const stringstream& s);
       const string& get_name ( );
       template <class T>
              void   load_object (T& t);
       int    get_array_size( );
       template <class T>
              void  load_array (T*& ptr);
};
```

Member Functions:

1. ***Deserializer(const stringstream &s)***

   The parameter is the stringstream to which the object was serialized.

2. ***const string& get_name ( );***

   It returns the class name if it was pushed at the Serializer end. Else it will return NULL.

3. ***template <class T>***
   void load_object (T& t);

   The parameter is the type in which serialized object will be restored.

4. ***int get_array_size( );***

   Returns the size of the saved array.

5. ***template <class T>***
   void load_array (T*& ptr);

   The parameter is the pointer where you want to restore the array.

SERIALIZATION TUTORIAL

Features supported
A simple serializable Class
Containment
Inheritance
Pointers
Arrays
Legacy Classes

1) **Features supported by the system**

- C++ primitive types
- User Defined Types
- Pointers of User Defined Types and Char*
- Arrays
- Self-Identifying Streams
- STL container and adapters
- Complex object graphs
- Inheritance

2) **A simple serializable Class**

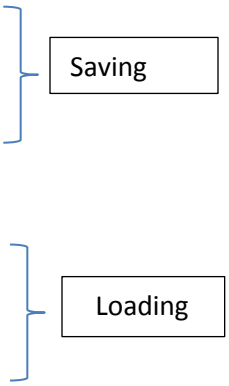
Any User Defined Types to be saved/loaded should implement two functions namely serialize and deserialize. Data members should be pushed using << operator and retrieved using >>. The sequence should be maintained. Add Serializer and Deserializer class as friend classes. This enables serialize and deserialize functions to be private.

To invoke serialization using our library, the user should include "serialization.h" in the code module.

For example,

```
Class Student{

      friend class Serializer;
      friend class Deserializer;

      string name;
      int  age;
      char grade;
```

```
        void serialize (Serializer &s){
              s<< name << age<< grade;
        }
        void deserialize (Deserializer &d){
              s>> name >> age >> grade;
        }
}

int main()
{
        Student st;
        Serializer ser;
        ser.save_object(st, "Student");
        string_stream str = ser.get_stream();

        /*Write to file / send over network*/

        Student st1;
        Deserializer dz(streamFromFile);
        string objName = dz.get_name( );
        /*Map "name" to object creation*/
        dz.load_object(st1);
}
```



- **To Save Objects:**
  void save_object(T & object, string class name);

  Call the function save_object and pass the object to be serialized as the parameter. You can optionally pass the class name as the second parameter. This optional parameter helps the library to generate named streams. The user can ask for the class name before deserializing. This way the user need not remember the order in which the objects are serialized.

- **To save the stream:**
  const stringstream get_stream();

  The object will be serialized as a stringstream. Call function get_stream() to get the stringstream. The user can save the stringstream in a file or send it across network.

- **To Load Object:**

  Deserializer dz(streamFromFile);

  It is mandatory to initialize Deserializer with the stringstream that was generated by the serializer.

  string get_name () ;

  Returns the name of the class. If the class name has been saved during serialization this function will return the same. If not it will return Null.

template <class T> void load_object (T& t);

Pass the object to be resurrected. The data members will be filled correctly in 't'.

3) **Containment:**

If the class has another object, just push that object like any other member. Our library will recursively call the serializer of the contained objects.

```
Class Circle
{
      Point p;
      double radius;
      void serialize(Serializer & sr){
            sr << p <<radius;
       }
      void deserialize(Deserializer &dz){
            dz >> p >>radius;
       }
 }
```

4) **Inheritance**

Derived class should include serialization of the base class.

```
Class Square : public Shape
{
      int side;
      void serialize (Serializer & sr) {
            sr << *(static_cast<shape *>(this)) <<side;
      }
      void deserialize(Deserializer &dz) {

      sr >> *(static_cast<shape *>(this)) >>side;
}
```

The derived class object should be type casted to base type and then pushed.

5) **Pointers**

To support custom memory management users can opt to provide a static function named *allocate_memory( ).* If this function is not provided , our Serializer library will allocate memory using the default new.

Eg: static tree_node* allocate_memory( );

No serialization support for pointers to primitive types except char *.

A sample class supporting custom memory management

```
Class  tree_node {
      friend class Serializer;
      friend class Deserializer;

      int         value;
      string      color;
```

```
        tree_node*  left;
        tree_node*  right;

        void serialize(Serializer &s){
              s<<value<<color<<left<<right;
        }
        void deserialize (Deserializer &d){
              d>>value>>color>>left>>right;
        }

        static tree_node * allocate_memory();
}

int main()
{
        tree_node * tree;
        Serializer ser;
        ser.save_object(tree, "tree_node");
        string_stream str = ser.get_stream();

        /*Write to file / send over network*/

        tree_node t1;
        Deserializer dz(streamFromFile);
        string objName = dz.get_name( );
        /*Map "name" to object creation*/
        dz.load_object(t1);
}
```

Saving

Loading

6) **Arrays**

Arrays and pointers are treated separately. We provide seperate interfaces namely save_array and load_array for storing and restoring arrays.

```
template <class T>
      void save_array(const T* ptr , int n, const string name =
                                                        "dummy");
template <class T>
      void load_array (T*& ptr);
int   get_array_size();
```

To save arrays the user should pass the length of the array as well. If the user just passes the pointer and not the length only the first element will be serialized.

While deserializing the user should first ask the length of the array using get_array_size, then call load_object passing the pointer as the parameter. The user should allocate enough memory to restore the array.

```
int main(){
      Student obj[10]; // fill later
      ser.save_array(obj, 10,  "Student");
      /* if the user calls save_object only obj[0] will be
      saved */
      Student* st1;
```

```
        Deserializer dz(stmFromFile);
        string objName = dz.get_name( );
        int length = get_array_size(); /*Returns the length of the array */
        st1 = new student [10];
        dz.load_array(st1);
    }
```

7) **Legacy Classes**

For legacy classes that cannot be modified the user has to write an Adaptor class and provide serialize and deserialize functions.

For example:

```
Class Legacy
{
      int val; string name;
public:

      int getVal();
      void setVal();
      string getName();
      void setName(string n);
};

Class Adaptor : public Legacy
{
     void serialize(Serializer& sz) {
         sz << getVal() << getName ();
     }
     void deserialize(Deserializer& dz) {
         int v; string n;
         dz >> v >> n;
         setVal(v); setName(n)
     }
}
```

Reference

1. Serialization http://en.wikipedia.org/wiki/Serialization
2. A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur.
   Cooperative task management without manual stack management.
   In Proceedings of the 2002 Usenix ATC, June 2002.
   http://citeseer.nj.nec.com/543557.html
3. Serialization faq http://www.parashift.com/c++-faq-lite/serialization.html
4. C++0x FAQ http://www2.research.att.com/~bs/C++0xFAQ.html#default
5. Clear- Objects http://www.clear-objects.com/Object-Serialization/
6. Reflection http://java.sun.com/developer/technicalArticles/ALT/Reflection/
7. Boost Serialization http://www.boost.org/doc/libs/1_46_1/libs/serialization/doc/tutorial.html