iJRASET

International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 2 **Issue:** X **Month of publication:** October 2014

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

Volume 2 Issue X, October 2014
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology(IJRASET)

Serialization and Objects with visual C++

Ankit Saxena[1], Himanshi Jhamb[2]
Information Technology Department, Dronacharya College Of Engineering, Gurgaon

***Abstract:** Serialization is the process of translating data structures or object state into a format that can be stored and reconstructed later in the same or another computer environment. When the resulting series of bits 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. Serialization of object-oriented objects does not include any of their associated methods with which they thee previously inextricably linked. This process of serializing an object is also called marshalling an object. A file represents a sequence of byte on the disk where a group of related data is stored. File is created for permanent storage of data. In this research paper, there is a detailed study about Serialization and File Handling with Visual C++.*

I. INTRODUCTION

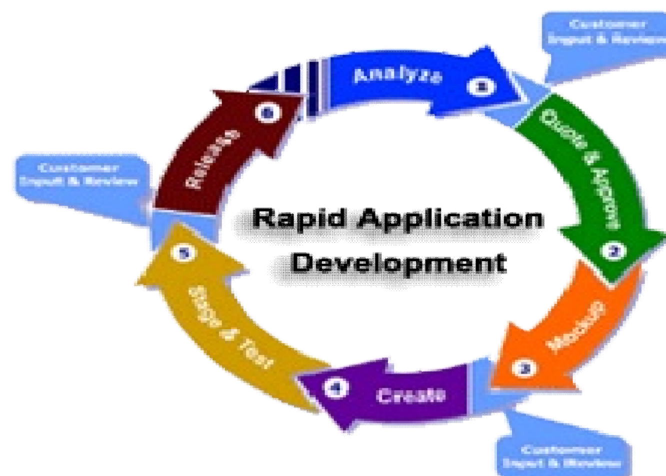
Serialization is the process of writing or reading an object to or from a persistent storage medium such as a disk file. Serialization is ideal for situations where it is desired to maintain the state of structured data during or after execution of a program. Using the serialization objects provided by MFC allows this to occur in a standard and consistent manner, relieving the user from the need to perform file operations by hand.

MFC supplies built-in support for serialization in the class CObject. Thus, all classes derived from CObject can take advantage of CObject's serialization protocol.

The basic idea of serialization is that an object should be able to write its current state, usually indicated by the value of its member variables, to persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from the storage. Serialization handles all the details of object pointers and circular references to objects that are used when we serialize an object. A key point is that the object itself is responsible for reading and writing its own state. Thus, for a class to be serializable, it must implement the basic serialization operations. As shown in the Serialization group of articles, it is easy to add this functionality to a class.

MFC uses an object of the CArchive class as an intermediary between the object to be serialized and the storage medium. This object is always associated with a CFile object, from which it obtains the necessary information for serialization, including the file name and whether the requested operation is a read or write. The object that performs a serialization operation can use the CArchive object without regard to the nature of the storage medium.

A CArchive object uses overloaded insertion (<<) and extraction (>>) operators to perform writing and reading operations. For more information, see Storing and Loading CObjects via an Archive in the article Serialization: Serializing an Object.



II. RELATED WORK

There are various operations performed in VC++ under serialization, some of them are given below:

A. Making a Serializable Class.

B. Serializing an Object.

All of the above operations are discussed and studied below.

A. Making a Serializable Class

There are five main steps which are required to make a VC++ class serializable. They are listed below and are explained in the following sections:

- Deriving the class from CObject.
- Overriding the serialize member function.
- Using the DECLARE_SERIAL macro in the class declaration.
- Defining the constructor that takes no argument.
- Using the IMPLEMENT_SERIAL macro in the implementation file of the class.

1) Deriving the class from object

The basic serialization protocol and functionality are defined in the CObject class. By deriving the class from CObject, as shown in the following declaration of class CPerson, we gain access to the serialization protocol and functionality of CObject**.**

2)Overriding the serialize member function

The Serialize member function, which is defined in the CObject class, is responsible for actually serializing the data necessary to capture an object's current state. The Serializefunction has a CArchive argument that it uses to read and write the object data. The CArchive object has a member function, IsStoring, which indicates whether Serialize is storing or loading. Using the results of IsStoring as a guide, we either insert the object's data in the CArchive object with the insertion operator (<<) or extract data with the extraction operator (>>).

Consider a class that is derived from CObject and has two new member variables, of types CStringand WORD. The following class declaration fragment shows the new member variables and the declaration for the overridden Serialize member function:

```
Class CPerson: public CObject

{

Public:

DECLARE_SERIAL(CPerson)

CPerson();

Virtual ~CPerson();

CString m_name;
```

```
WORD m_number;

Void Serialize(CArchive& archive);

};
```

a. To override the serial member function

- Call the base class version of Serialize to make sure that the inherited portion of the object is serialized.
- Insert or extract the member variables specific to the class.

The insertion and extraction operators interact with the archive class to read and write the data. The following example shows how to implement Serialize for the CPerson class declared above:

```
Void CPerson :: Serialize (CArchive& archive)

{

CObject :: serialize (archive);

If ( archive.IsStoring())
archive<<  m_name  << m_number;
else>> m_name >> m_number;
}
```

3)Using the declared macro in a class

The DECLARE_SERIAL macro is required in the declaration of classes that will support serialization, as shown here:

```
class CPerson : public CObject
{
Public:
DECLARE_SERIAL ()
};
```

4) Defining the constructor that takes no argument

MFC requires a default constructor when it re-creates the objects as they are deserialized. The deserialization process will fill in all member variables with the values required to re-create the object.
This constructor can be declared public, protected, or private. If we make it protected or private, we help make sure that it will only be used by the serialization functions. The constructor must put the object in a state that allows it to be deleted if necessary.

5) Using the IMPLEMENT_SERIAL macro in the implementation file of the class

The IMPLEMENT_SERIAL macro is used to define the various functions needed when we derive a serializable class from CObject. We use this macro in the implementation file (.CPP) for the class. The first two arguments to the macro are the name of the class and the name of its immediate base class.
The third argument to this macro is a schema number. The schema number is essentially a version number for objects of the class. Use an integer greater than or equal to 0 for the schema number.
The MFC serialization code checks the schema number when reading objects into memory. If the schema number of the object on disk does not match the schema number of the class in memory, the library will throw a CArchiveException, preventing the program from reading an incorrect version of the object.

International Journal for Research in Applied Science & Engineering Technology(IJRASET)

If we want the Serialize member function to be able to read multiple versions — that is, files written with different versions of the application — we can use the valueVERSIONABLE_SCHEMA as an argument to the IMPLEMENT_SERIAL macro. For usage information and an example, see the GetObjectSchema member function of classCArchive.
The following example shows how to use IMPLEMENT_SERIAL for a class, CPerson, which is derived from CObject:

IMPLEMENT_SERIAL (CPerson, CObject, 1)

B. Serializing An Object

Making a Serializable Class shows how to make a class serializable. Once we have a serializable class, we can serialize objects of that class to and from a file via a CArchive object.

A CArchive object provides a type-safe buffering mechanism for writing or reading serializable objects to or from a CFile object. Usually the CFile object represents a disk file; however, it can also be a memory file, perhaps representing the Clipboard.
A given CArchive object either stores data or loads data, but never both. The life of a CArchive object is limited to one pass through writing objects to a file or reading objects from a file. Thus, two successively created CArchive objects are required to serialize data to a file and then deserialize it back from the file.
When an archive stores objects to a file, the archive attaches the CRuntimeClass name to the objects. Then, when another archive loads objects from a file to memory, theObject-derived objects are dynamically reconstructed based on the CRuntimeClass of the objects. A given object may be referenced more than once as it is written to the file by the storing archive. The loading archive, however, will reconstruct the object only once.
As data is serialized to an archive, the archive accumulates the data until its buffer is full. Then the archive writes its buffer to the CFile object pointed to by the CArchive object. Similarly, as we read data from an archive, it reads data from the file to its buffer and then from the buffer to the deserialized object. This buffering reduces the number of times a hard disk is physically read, thus improving the application's performance.

Storing and loading CObjects via an archive requires extra consideration. In certain cases, you should call the Serialize function of the object, where the CArchive object is a parameter of the Serialize call, as opposed to using the << or >> operator of the CArchive. The important fact to keep in mind is that the CArchive >> operator constructs the CObject in memory based on CRuntimeClass information previously written to the file by the storing archive.
Therefore, whether you use the CArchive << and >> operators, versus calling Serialize, depends on whether you need the loading archive to dynamically reconstruct the object based on previously stored CRuntimeClass information. Use the Serialize function in the following cases:

- When deserializing the object, you know the exact class of the object beforehand.
- When deserializing the object, you already have memory allocated for it.

REFERENCES

[1] Microsoft Visual C++ by Steven Holzner.
[2] Visual C++ programming, 2nd edition by Steven Holzner.
[3] Using Visual C++ Basic for application by Paul Sanna.
[4] Visual Basic Programming by Steven Holzner.
[5] Visual C++ from the ground up by mucller.
[6] Programming Visual C++ by David J. Kruglinski.
[7] MSDN.Microsoft.com
[8] Rapid Application Development by James Martin.

10.22214/IJRASET

45.98

IMPACT FACTOR:
7.129

IMPACT FACTOR:
7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY