

Joe Student

CET421

LAB 1: UNIX™ File Services

January 1, 1980

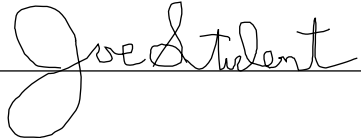
Due Week 16

For: Sr. Professor Wheeler

OPERATIONAL SIGN OFF \_\_\_\_\_

FINAL SIGN OFF \_\_\_\_\_

"THIS IS THE ORIGINAL WORK OF JOE STUDENT"

  
\_\_\_\_\_

## Introduction:

This project demonstrates serialization using traditional Unix-style file input and output by creating a simple database, saving it to disk, and then recalling it. Serialization is the process of moving objects from one place to another. In the case of this program, they are moved from memory to disk, and then back to memory. The program demonstrates that serialization is not a complex process and in fact the Microsoft mechanisms are probably overkill.

## Theory of Operation:

CLEARLY MARK each section of the report.

The program consists of three classes, `CDatabase`, `CDatabaseEntry`, and `CSerializer`. Most of the action occurs within the code for these classes. Each `CDatabaseEntry` object holds *one* entry of the database, and a `CDatabase` object holds 100 `CDatabaseEntry` objects. (The value 100 is hard-coded). The methods `SetData()` and `Print()` are used to set and print data values for each `CDatabaseEntry` object.

When the program begins execution, the following activities take place:

- A `CDatabase` object called "db" is created.
- The `Initialize()` method of the `CDatabase` object is used to fill it with dummy data.
- The contents of the database object "db" are printed using the `Dump()` method.
- The database is serialized to disk using the `SerializeTo()` method in `CDatabase`. The `SerializeTo()` method relies on the `Save()` method of the `CSerializer` class.
- The `CSerializer::Save()` method automatically calls the `WriteData()` method of each `CDatabaseEntry` object within the database. In turn, each `CDatabaseEntry` object responds to this message by calling the `WriteData()` method of `CSerializer`, which results in the specific data for each object being written to the disk file.
- A second, blank database is created as "db2."
- Object "db2" is serialized from the disk file created from "db1."
- To verify that serialization worked, the contents are "db2" are dumped to the screen.

Serialization is accomplished within the `CSerializer` helper class by using the `fread()` and `fwrite()` library functions. For example, when a `CDatabase` object is requested to serialize itself to disk, the `SerializeTo()` method is called. Within this method, the following actions take place:

- A disk file is created in binary write ("wb") mode. (This *must* be done in binary mode to assure that no character translations occur, which would interfere with the binary data being serialized).
- If file opening fails (we get back NULL for the file handle), a (-1) value is returned to alert the caller.
- The static `Save()` method of `CSerializer` is called, passing the open file handle, a pointer to the array of objects to write, and the number of objects to write (hard coded to 100 in this program.)
- `CSerializer::Save()` iterates through each object in the array, sending it a message to write itself by calling the method `WriteData()` that *must* be implemented by each object.
- The `WriteData()` method of each object calls the `CSerializer::WriteData()` method as many times as needed to write all of its data members to the stream. In this example, two data members from each `DatabaseEntry` object are written, `m_szData` and `m_nSerial`.
- The write file is closed.
- A test is made to see if 100 objects actually got written, as reported by the return value of `CSerializer::Save()`. If not, an error value of (-1) is returned; otherwise, (0) is returned to indicate success.

Deserialization occurs in a similar fashion as shown in the `SerializeFrom()` method:

- A disk file is created in binary read ("rb") mode. (This *must* be done in binary mode to assure that no character translations occur, which would interfere with the binary data being serialized).
- If file opening fails (we get back NULL for the file handle), a (-1) value is returned to alert the caller.
- The static `Load()` method of `CSerializer` is used to read 100 `DatabaseEntry` objects into the internal array "`m_pArray`."
- The disk file is closed.
- The status reported by `Load()` is checked; if 100 objects weren't read, an error value of (-1) is reported. Otherwise, (0) is returned to indicate success.

## Program Listings

```
// Database.cpp: implementation of the CDatabase class.
// Author: Student, Joe
// Version: 1.0 (January 1, 1980)
// Revision History: NONE
//////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Database.h"

//////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////

CDatabase::CDatabase()
{

    m_pArray = new CDatabaseEntry[100];

}

CDatabase::~CDatabase()
{

    delete[] m_pArray;

}

//////////////////////////////////////////////////////////////////
// Method: Initialize
// Purpose: Loads dummy data into each entry in the database for testing
// Parameters: None
// Returns: None
//////////////////////////////////////////////////////////////////

void CDatabase::Initialize()
{
    int i;
    char buf[128];

    for(i=0;i<100;i++)
    {
        sprintf(buf, "Test data for item # %d", i+1 );
        m_pArray[i].SetData(buf, i*i );
    }

}

//////////////////////////////////////////////////////////////////
// Method: Dump
// Purpose: "Dumps" the entire database to the screen for inspection.
// Parameters: None
// Returns: None
//////////////////////////////////////////////////////////////////

void CDatabase::Dump()
{
    int i;

    for(i=0;i<100;i++)
        m_pArray[i].Print();

}
```

YOUR NAME is  
required on all listings.

All functions other  
than default  
constructors must have  
detailed header.

```

////////////////////////////////////
// Method: SerializeTo
// Purpose: Writes this object to a disk or network file.
// Parameters: szFile, a character string giving the full pathname.
// Returns: 0 on success, or -1 on failure.
////////////////////////////////////

int CDatabase::SerializeTo(char *szFile)
{
FILE* f1;
int nResult;

f1 = fopen(szFile, "wb");

if (f1 == NULL) return -1;

nResult = CSerializer::Save(f1, // Stream to serialize to
                             m_pArray, // What to serialize (an
                             array of objects)
                             100); // Number of objects to
serialize

fclose(f1);

if (nResult != 100) return -1;

return 0;
}

////////////////////////////////////
// Method: SerializeFrom
// Purpose: Reads this object from a disk or network file.
// Prequisite: This object must FIRST be constructed using any available
//              constructor or class factory mechanism.
// Parameters: szFile, a character string giving the full pathname.
// Returns: 0 on success, or -1 on failure.
////////////////////////////////////

int CDatabase::SerializeFrom(char *szFile)
{
FILE* f1;
int nResult;

f1 = fopen(szFile, "rb");

if (f1 == NULL) return -1;

nResult = CSerializer::Load(f1,m_pArray,100);

fclose(f1);

if (nResult != 100) return -1;

return 0;
}

```

```

////////////////////////////////////
// DatabaseEntry.cpp: implementation of the CDatabaseEntry class.
// Author: Student, Joe
// Version: 1.0 (January 1, 1980)
// Revision History: NONE
////////////////////////////////////

#include "stdafx.h"
#include "DatabaseEntry.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

CDatabaseEntry::CDatabaseEntry()
{
    m_nSerial = 0;
    m_szData[0]=0;
}

CDatabaseEntry::~CDatabaseEntry()
{
}

CDatabaseEntry::CDatabaseEntry(char * name, int x)
{
    m_nSerial = x;
    strcpy(m_szData, name);
}

////////////////////////////////////
// Method: Print
// Purpose: Prints the object's contents to the screen
// Parameters: None
// Returns: None
////////////////////////////////////

void CDatabaseEntry::Print()
{
    printf("ID: %d\nName: %s\n", m_nSerial, m_szData );
}

////////////////////////////////////
// Method: SetData
// Purpose: Sets the data fields of this object to the specified values.
// Parameters: szName, "Name" field value; nID, "serial number" value
// Returns: None
////////////////////////////////////

void CDatabaseEntry::SetData(char *szName, int nID)
{
    strcpy(m_szData, szName);
    m_nSerial = nID;
}

```

```

////////////////////////////////////
//
// ReadData()
//
// Function: This method calls the method of the parent class to
// read particular object data from the specified stream.
//
// The implementor of this method MUST specify the specific data
// members of the object that are to be serialized by adding
// calls to CSerializer::ReadData() for each member, as shown below.
//
// Returns: 1 on success, 0 on error.
//
////////////////////////////////////

int CDatabaseEntry::ReadData(FILE *hStream)
{

if (0==CSerializer::ReadData(hStream,&m_nSerial,sizeof(m_nSerial)))
    return 0;

if (0==CSerializer::ReadData(hStream,m_szData,sizeof(m_szData)))
    return 0;
    else
        return 1;    // success

}

////////////////////////////////////
//
// WriteData()
//
// Function: This method calls the method of the parent class to
// write particular object data to the specified stream.
//
// The implementor of this method MUST specify the specific data
// members of the object that are to be serialized by adding
// calls to CSerializer::WriteData() for each member, as shown below.
//
// Returns: 1 on success, 0 on error.
//
////////////////////////////////////

int CDatabaseEntry::WriteData(FILE *hStream)
{

if (0==CSerializer::WriteData(hStream,&m_nSerial,sizeof(m_nSerial)))
    return 0;

if (0==CSerializer::WriteData(hStream,m_szData,sizeof(m_szData)))
    return 0;
    else
        return 1;    // success

}

```

## Conclusion

Serialization using conventional UNIX file I/O methods is very easy and reliable. However, there are several issues with this particular program:

- It always dumps 100 objects to disk, regardless of how many actually contain data. This is wasteful of disk space.
- There's nothing in each serialization stream that specifies how many objects are being written, or the version of each object. To properly implement a variable document size, additional information would need to be included in the file header to specify how much was being written in each file, along with versioning information.
- The `CSerializer` mechanism used requires that some serialization code be embedded into each type of serializable object. There's no mechanism for object introspection in C++, so there's no simple way around this issue.