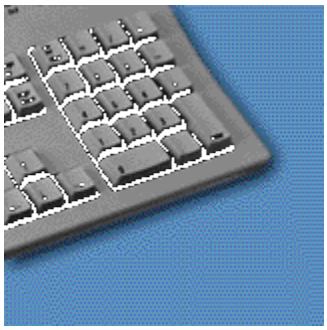


13. 직렬화 (Serialization)

<장도비라>
이 장은 7장에서 처음 작성한 Single 프로젝트 소스 분석의 마지막 장이다. 분석을 남겨둔 클래스는 CSingleView와 CSingleDoc인데 직렬화를 제외하고는 이전의 장들에서 모두 설명된 내용이다. 그래서 이 장에서는 직렬화의 분석에 집중한다. 우리는 MFC의 직렬화의 원리를 이해하기 위해 CArchive를 이용하는 단계별 예를 구현한다. 그리고, 이를 실제 MFC에서 구현해 본다. 마지막으로 다큐먼트,뷰와 프레임의 메시지 맵 순서를 확인함으로써 MFC의 원리 분석을 마친다.
</장도비라>

이제 원리 분석의 마지막 장이다. 이 장에서는 여러 객체들에 흩어져 있는 데이터를 차례대로(serial) 저장 - 그곳이 어느 곳이든 - 하거나 읽어 들이는 직렬화(serialization)에 대해 알아보고, 7장에서 처음 작성했던 Single 프로젝트의 다큐먼트와 뷰 클래스의 소스를 분석한다.

필자는 독자들이 MFC의 CString과 CFile 클래스의 사용법을 어느 정도 알고 있다고 가정한다. 그렇지 못한 독자들은 먼저 MSDN을 참고하여 CString과 CFile 클래스의 사용법을 익히기 바란다.

이 장에서 다음 내용들을 살펴볼 것이다.

- 직렬화의 의미
- 직렬화를 지원하기 위해 클래스에 매크로를 추가하는 법
- 직렬화를 위한 CArchive 클래스의 역할
- 직렬화를 위해 Serialize()멤버 함수를 작성하는 법
- Single 프로젝트의 나머지 소스 분석
- OnDraw()의 실제 동작
- 다큐먼트, 뷰와 프레임에 메시지가 맵되는 순서

MFC의 직렬화

<절도비라>

이 절에서는 일반적인 직렬화의 의미와 MFC의 직렬화를 살펴보고, 구체적인 MFC 직렬화 코드를 작성한다. 또한, CArchive가 직렬화의 다리(bridge)로 이용됨을 이해한다.

</절도비라>

MFC에서 **직렬화**란 일반적으로 하드 디스크 같은 영속적인 저장 장치(persistent storage device)에 여러 객체들을 차례대로 쓰거나 읽는 과정이다. 하지만 실제 직렬화란 여러 곳에 흩어져 있는 자료들을 "차례대로" 만드는 일반적인 방법을 가리키는 말이다.

예를 들면, 프로그램에서 10종류 클래스의 각 10개의 객체들, 총 100개의 객체들이 유지된다고 가정해 보자. 이러한 객체들의 상태를 프로그램이 종료한 후 다시 프로그램을 시작할 때 유지하고자 한다면, 어떠한 방법으로 객체들을 디스크 등에 저장하는 것이 필요해 진다. 그렇게 하기 위해서는 100개의 객체들을 "차례대로(serial)" 디스크에 저장해야 한다. 이러한 과정을 직렬화라고 한다. 직렬화가 어떻게 "차례대로"를 구현해야 하는지에 대해서는 관여하고 있지 않으며, "어느 곳"에 저장하는지도 관여하지 않음에 주목하라. 필자는 이 장에서 "어느 곳"을 "파일"에만 한정 시킬 것이다(실제로는 네트웍 스트림, 파일, 메모리 등 아무 것이나 될 수 있다).

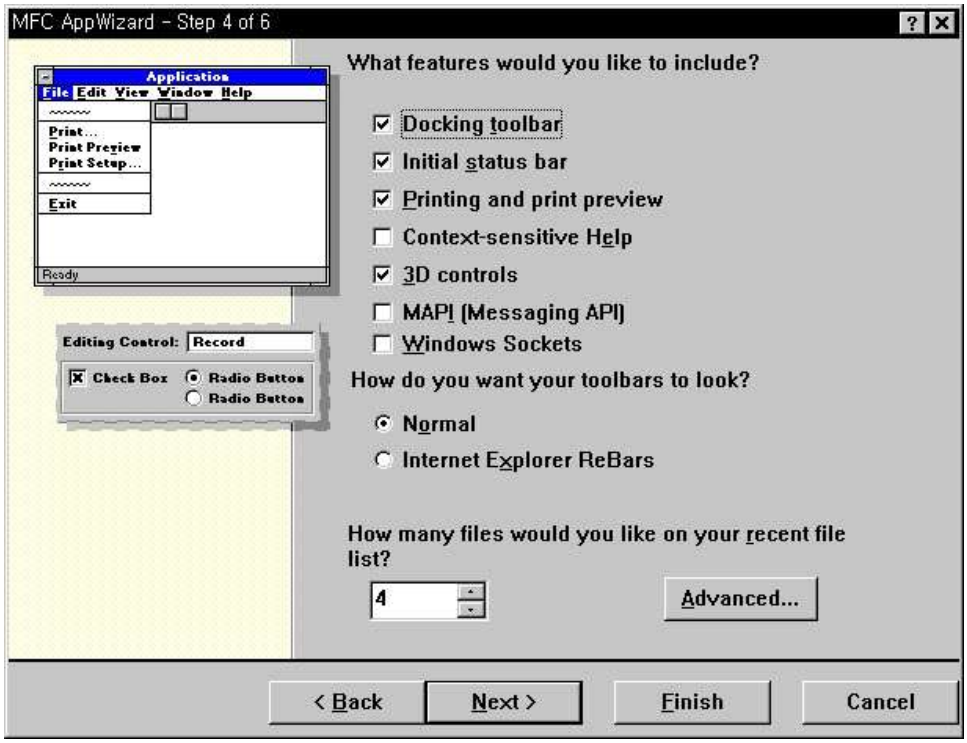
MFC의 구조에서 직렬화는 파일 입출력이나 네트웍 소켓으로 구현되지만, 숨겨진 입출력에 신경 쓰지 않고, 객체 자체의 읽고 쓰기에만 집중한다는 면에서 일반적인 입출력과는 다르다(일반적으로 직렬화는 이러한 방식으로 구현된다).

MFC 클래스에 직렬화를 지원하기 위해서는, 객체의 동적 생성과 직렬화에 관한 매크로와 Serialize()란 가상 함수를 구현해 주어야 한다. 먼저 MFC의 직렬화에 관한 원리를 이해하기 위해 간단한 예를 만들어 보자.

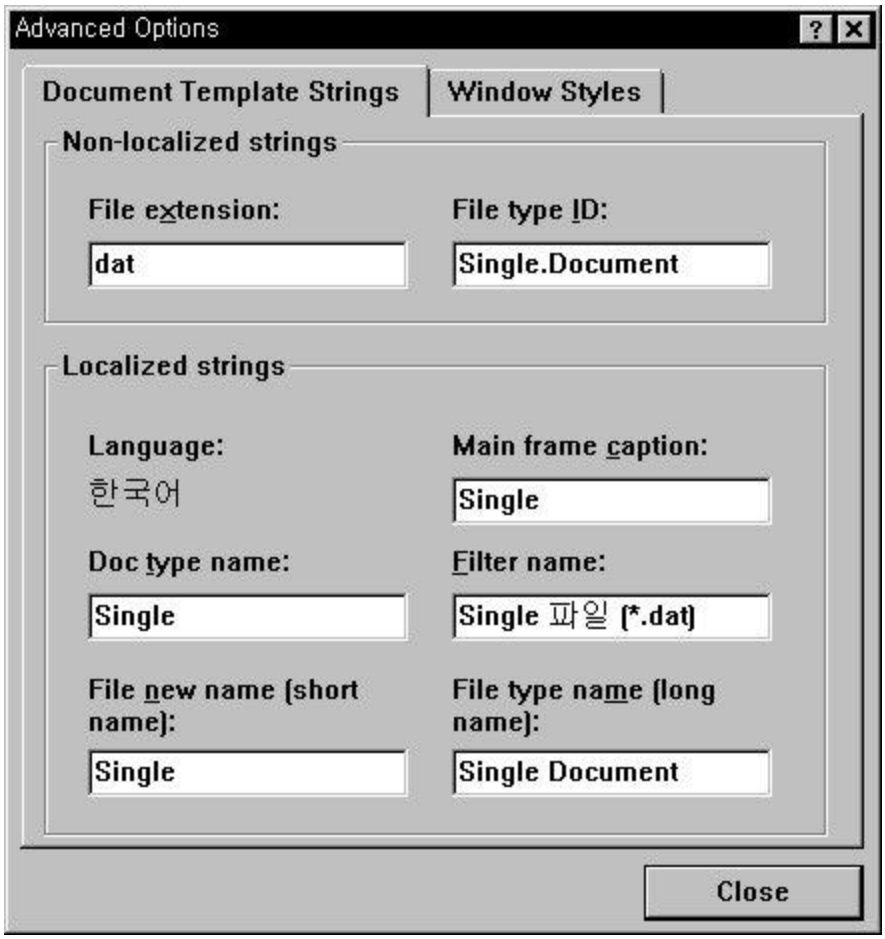
먼저 프로젝트 이름을 Single로 하여 MFC Single Document 응용 프로그램을 만든다. 이 때 단계 4에서 Advanced를 선택하여 File Extension에

```
dat
```

를 입력하도록 하자. 이 과정은 파일 확장자가 dat인 파일의 입출력을 직렬화 코드에 연결시킨다.

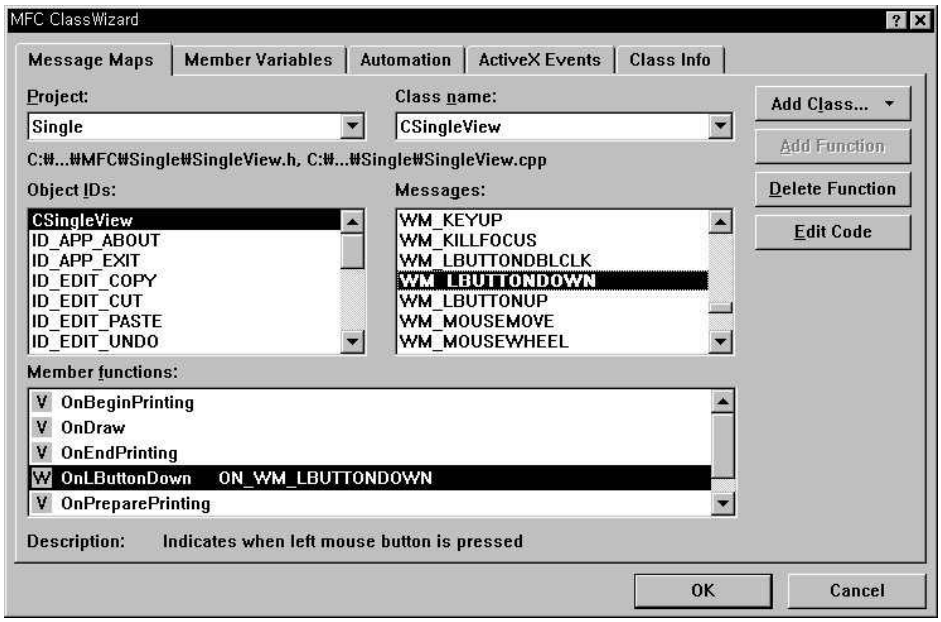


[그림 13.1] 단계 4: 단계 4에서 Advanced 버튼을 선택한다.



[그림 13.2] 파일 확장자 입력: File extension 박스에 dat라고 입력한다.

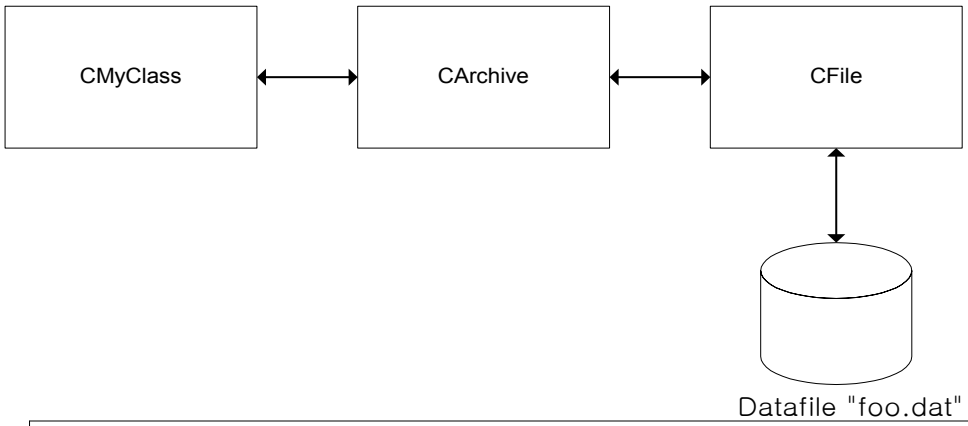
AppWizard가 코드 생성을 마치면, Insert→New Class...를 선택하여, 일반 클래스(Generic Class) CFoo를 만든다. 그리고 클래스 위저드를 실행하여 CSingleView에 WM_LBUTTONDOW 메시지를 맵한다.



[그림 13.3] WM_LBUTTONDOWN맵: 클래스 위저드를 실행하여 CSingleView의 WM_LBUTTONDOWN을 맵한다.

직렬화를 위한 다리, CArchive

직렬화에서 구체적인 파일 동작을 숨기기 위해 추상적인 인터페이스가 필요하다. 직렬화에서 파일 입출력을 위한 다리(bridge)는 CArchive이다. CArchive 객체를 통해 필요한 내용을 저장하거나 읽어 들인다.



[그림 13.4] 직렬화의 브리지 패턴: MFC의 직렬화는 동적으로 만들어진 CArchive 객체를 통해 파일에 접근한다. 직렬화의 구현자는 CMyClass의 직렬화 함수에만 집중한다.

MFC에 포함된 CArchive 클래스의 헤더 [예제 13.1]과 같다.

[예제 13.1] class CArchive

```
class CArchive
{
public:
// Flag values
    enum Mode { store = 0, load = 1, bNoFlushOnDelete = 2,
                bNoByteSwap = 4 };

    CArchive(CFile* pFile, UINT nMode, int nBufSize = 4096,
             void* lpBuf = NULL);
    ~CArchive();

// Attributes
    BOOL IsLoading() const; // (1)
    BOOL IsStoring() const;
    BOOL IsByteSwapping() const;
    BOOL IsBufferEmpty() const;

    CFile* GetFile() const;
    UINT GetObjectSchema(); // only valid when reading a CObject*
    void SetObjectSchema(UINT nSchema);

    // pointer to document being serialized -- must set to serialize
```

```
    //  COleClientItems in a document!
    CDocument* m_pDocument;

// Operations
    UINT Read(void* lpBuf, UINT nMax); // (2)
    void Write(const void* lpBuf, UINT nMax);
    void Flush();
    void Close();
    void Abort();   // close and shutdown without exceptions

    // reading and writing strings
    void WriteString(LPCTSTR lpsz);
    LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
    BOOL ReadString(CString& rString);

public:
    // Object I/O is pointer based to avoid added construction overhead.
    // Use the Serialize member function directly for embedded objects.
    friend CArchive& AFXAPI operator<<(CArchive& ar,
            const CObject* pOb);

    friend CArchive& AFXAPI operator>>(CArchive& ar, CObject*& pOb);
    friend CArchive& AFXAPI operator>>(CArchive& ar,
            const CObject*& pOb);

    // insertion operations
    CArchive& operator<<(BYTE by); // (3)
    CArchive& operator<<(WORD w);
    CArchive& operator<<(LONG l);
    CArchive& operator<<(DWORD dw);
    CArchive& operator<<(float f);
    CArchive& operator<<(double d);

    CArchive& operator<<(int i);
    CArchive& operator<<(short w);
    CArchive& operator<<(char ch);
    CArchive& operator<<(unsigned u);

    // extraction operations
    CArchive& operator>>(BYTE& by);
    CArchive& operator>>(WORD& w);
    CArchive& operator>>(DWORD& dw);
    CArchive& operator>>(LONG& l);
```

```
    CArchive& operator>>(float& f);
    CArchive& operator>>(double& d);

    CArchive& operator>>(int& i);
    CArchive& operator>>(short& w);
    CArchive& operator>>(char& ch);
    CArchive& operator>>(unsigned& u);

    // object read/write
    CObject* ReadObject(const CRuntimeClass* pClass);
    void WriteObject(const CObject* pOb);
    // advanced object mapping (used for forced references)
    void MapObject(const CObject* pOb);

    // advanced versioning support
    void WriteClass(const CRuntimeClass* pClassRef);
    CRuntimeClass* ReadClass(const CRuntimeClass*
        pClassRefRequested = NULL,
        UINT* pSchema = NULL, DWORD* pObTag = NULL);
    void SerializeClass(const CRuntimeClass* pClassRef);

    // advanced operations (used when storing/loading many objects)
    void SetStoreParams(UINT nHashSize = 2053, UINT nBlockSize = 128);
    void SetLoadParams(UINT nGrowBy = 1024);

// Implementation
public:
    BOOL m_bForceFlat;  // for COleClientItem implementation
                        // (default TRUE)
    BOOL m_bDirectBuffer;   // TRUE if m_pFile supports direct buffering
    void FillBuffer(UINT nBytesNeeded);
    void CheckCount();  // throw exception if m_nMapCount is too large

    // special functions for reading and writing (16-bit compatible)
    // counts
    DWORD ReadCount();
    void WriteCount(DWORD dwCount);

    // public for advanced use
    UINT m_nObjectSchema;
    CString m_strFileName;

protected:
```

```
    // archive objects cannot be copied or assigned
    CArchive(const CArchive& arSrc);
    void operator=(const CArchive& arSrc);

    BOOL m_nMode;
    BOOL m_bUserBuf;
    int m_nBufSize;
    CFile* m_pFile;
    BYTE* m_lpBufCur; // (4)
    BYTE* m_lpBufMax;
    BYTE* m_lpBufStart;

    // array/map for CObject* and CRuntimeClass* load/store
    UINT m_nMapCount;
    union
    {
        CPtrArray* m_pLoadArray;
        CMapPtrToPtr* m_pStoreMap;
    };
    // map to keep track of mismatched schemas
    CMapPtrToPtr* m_pSchemaMap;

    // advanced parameters (controls performance with large archives)
    UINT m_nGrowSize;
    UINT m_nHashSize;
};
```

소스를 보면 (1) 로딩중인지 저장중인지를 얻어내는 헬퍼(helper), (2) 직접적인 입출력 멤버 함수들, (3) 기본 타입의 입출력을 위한 오버로딩된 operator<<(), operator>>() 함수들, (4) 입출력 버퍼링(buffering)을 위한 멤버들이 존재함을 알 수 있다.

이제 CArchive의 동작을 확인하기 위해, 왼쪽 버튼을 눌렀을 때, foo.dat 파일을 만들고 CArchive 객체를 이용하여, 이를 접근하는 것을 구현해 보자. 이 예에서 클래스를 직렬화하기 위한 MFC의 매크로 등을 사용하지는 않을 것이다.

CSingleView의 OnLButtonDown()의 몸체를 아래와 같이 작성한다.

단계 1: 파일을 연다.

오브젝트를 foo.dat파일로 직렬화하려면, 먼저 파일을 적당한 모드로 연다.

```
// step 1: Open data file "foo.dat"
CFile* pFile = new CFile();
ASSERT (pFile != NULL);
if ( !pFile->Open("foo.dat", CFile::modeWrite|CFile::modeCreate) )
{
    // Handle error
    return;
}//if
```

단계 2: CArchive 객체를 만든다.

그리고, CFile의 대리자 CArchive 객체를 만들고, 만들어진 CFile객체를 연결한다.

```
// step 2: Create archive ...
bool bReading = false;  // ... for writing
CArchive* pArchive = NULL;
try
{
    pFile->SeekToBegin();
    UINT uMode = (bReading ? CArchive::load : CArchive::store);
    pArchive = new CArchive(pFile, uMode);
    ASSERT (pArchive != NULL);
}//try
catch (CException* pException)
{
    // Handle error
    return;
}//catch
```

단계 3: 객체의 직렬화 멤버 함수를 호출한다.

이제 직렬화를 원하는 객체의 직렬화 멤버 함수 Serialize()를 호출할 수 있다.

```
// step 3: call Serialize()
```

```
CFoo*   pFoo = new CFoo();
pFoo->Serialize( *pArchive );
delete pFoo;
```

CFoo에는 반드시 Serialize()가 가상 함수로 존재해야 한다는 것을 주의하자. 예에서는 그렇지 않지만 미리 구현된 MFC의 직렬화 로직(logic)이 가상 함수를 호출하기 때문이다.

단계 4: 다리(bridge) 객체와 파일 객체를 파괴한다.

```
// step 4: clean up
pArchive->Close();
delete pArchive;
pFile->Close();
delete pFile;
```

위의 네 단계에서 단계 1,2와 단계 4는 이미 MFC에 구현되어 있다. 그러므로 MFC 응용 프로그램에서 직렬화를 위해서는 단계 2만을 구현하면 되는 것이다. 즉 Serialize()를 오버라이드한다. OnLButtonDown()의 소스는 아래 [예제 13.2]와 같다.

[예제 13.2] OnLButtonDown()

```
void CSingleView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // step 1: Open data file "foo.dat"
    CFile* pFile = new CFile();
    ASSERT (pFile != NULL);
    if ( !pFile->Open("foo.dat", CFile::modeWrite|CFile::modeCreate) )
    {
        // Handle error
        return;
    }//if

    // step 2: Create archive ...
    bool bReading = false;  // ... for writing
    CArchive* pArchive = NULL;
    try
    {
```

```
        pFile->SeekToBegin();
        UINT uMode = (bReading ? CArchive::load : CArchive::store);
        pArchive = new CArchive(pFile, uMode);
        ASSERT (pArchive != NULL);
    }//try
    catch (CException* pException)
    {
        // Handle error
        return;
    }//catch

    // step 3: call Serialize()
    CFoo*   pFoo = new CFoo();
    pFoo->Serialize( *pArchive );
    delete pFoo;

    // step 4: clean up
    pArchive->Close();
    delete pArchive;
    pFile->Close();
    delete pFile;

        CView::OnLButtonDown(nFlags, point);
}
```

이제 직렬화를 지원하는 CFoo클래스를 완성해 보자. CFoo클래스는 Serialize()를 가상 함수로 반드시 구현해야 한다. 먼저 헤더에 Serialize() 선언을 포함한다.

```
class CFoo
{
    // ...
    virtual void Serialize(CArchive& ar);
};//class CFoo
```

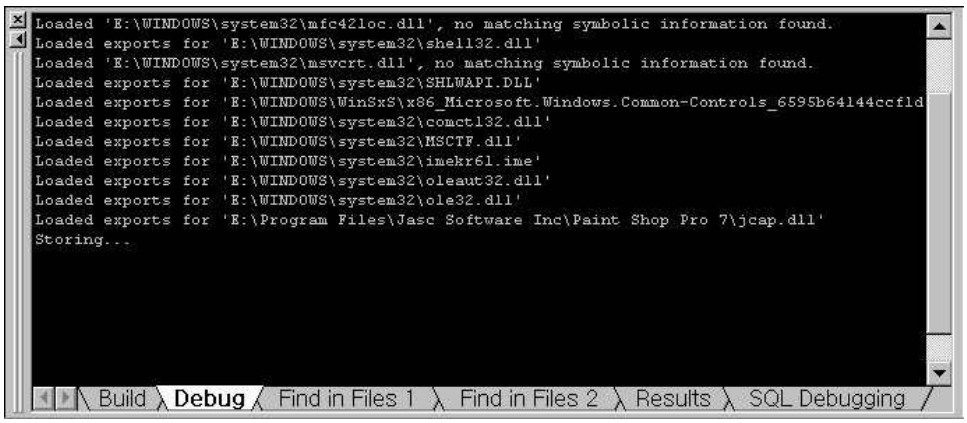
그리고, 구현 파일에 함수를 구현한다.

```
void CFoo::Serialize(CArchive& ar)
{
    if ( ar.IsStoring() )
    {
```

```
        AfxTrace( "Storing...\n" );
    }
    else
    {
        AfxTrace( "Loading...\n" );
    }//if.. else..
}//CFoo::Serialize()
```

간단하게 비주얼 C++의 디버그 창에 문자열을 출력하도록 구현했다. 하지만, CArchive가 모든 기본 타입에 대해 오버로드된 operator<<()와 operator>>()를 제공하는 것을 기억하자. 저장을 위해서는 operator<<()를, 읽어오기를 위해서는 operator>>()를 사용할 수 있다.

F5를 눌러 프로그램을 디버그 런(Debug Run)해서 왼쪽 버튼을 누르면 디버그 창에 문자열이 출력되는 것을 [그림 13.5]처럼 확인해 볼 수 있다.



[그림 13.5] CFoo의 Serialize(): 디버그 창에 직렬화 동작을 출력한다.

자 그럼 MFC의 특정 클래스가 직렬화를 지원하기 위해서는 무엇이 필요한가? 단지 Serialize()란 이름의 가상 함수를 제공하기만 하면 된다. 하지만, 이 가상함수가 제대로 동작해야 하므로, 자신이 만드는 클래스를 반드시 CObject에서 상속받아야 한다. 아래는 MFC 소스에 포함된 CObject 선언의 일부인데, 제공되는 네 가지 가상함수 중에 Serialize()가 포함된 것을 알 수 있다.

```
class CObject
{
public:
```

```
    virtual CRuntimeClass* GetRuntimeClass() const;
    virtual void Serialize(CArchive& ar);
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
};
```

다큐먼트 클래스에는 이미 이 조건을 만족하므로 Serialize() 멤버 함수를 구현해 주기만 하면 되는 것이다.

하지만, 실제로 MFC의 직렬화는 추가적인 두 가지의 작업이 더 필요한데, 하나는 클래스를 동적 생성이 가능하도록 해 주는 것이고, 다른 하나는 저장된 파일에서 클래스 이름으로 객체를 구분하기 위한 프렌드 함수 operator>>()를 구현해 주는 것이다. 첫 번째 작업은 CArchive의 내부 동작 때문에 필요한 것이고, 두 번째 작업은 오브젝트가 바이너리로 저장된 경우, 읽어 들이는 작업을 할 때 오브젝트를 구분하기 위해서 필요한 것이다.

MFC는 이러한 추가적인 작업에 대한 매크로를 제공한다. 매크로는 아래와 같다(AFX.H의 780번 줄)□. DECLARE_SERIAL()은 헤더에 필요한 매크로이

<여기서 잠깐>
MFC 소스를 찾을 때는 찾기를 원하는 문자열 위에서 F12를 누른다.
</여기서 잠깐>

고, IMPLEMENT_SERIAL()은 구현 파일에 필요한 매크로이다.

```
#define DECLARE_SERIAL(class_name) \
    _DECLARE_DYNCREATE(class_name) \
    AFX_API friend CArchive& AFXAPI operator>>(CArchive& ar,\
 class_name* &pOb);

///...

#define IMPLEMENT_SERIAL(class_name, base_class_name, wSchema) \
    CObject* PASCAL class_name::CreateObject() \
        { return new class_name; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
        class_name::CreateObject) \
    AFX_CLASSINIT _init_##class_name(RUNTIME_CLASS(class_name)); \
```

```
    CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) \
        { pOb = (class_name*) ar.ReadObject(\
RUNTIME_CLASS(class_name)); \
            return ar; } \
```

그러므로 만드는 MFC 클래스 CTest를 직렬화를 지원하게 만들려면 먼저 CTest의 헤더에 아래의 문장을 포함한다.

```
class CTest : public CObject
{
protected:
    DECLARE_SERIAL(CTest)
public:
    virtual void Serialize(CArchive& ar);

    // ...
};//class CTest
```

그리고 CTest의 구현 파일에 다음을 포함하고, Serialize()를 구현한다.

```
    IMPLEMENT_SERIAL(CTest,CObject,1)
```

직렬화를 지원하기 위한 클래스는 CObject의 자손(descendent) 중 누구도 될 수 있다. 만약 CTest가 MFC의 CByteArray의 기능을 이용한다면

```
class CTest : public CByteArray
{
    // ...
};
```

처럼 작성할 수 있는데, 이것은 CByteArray의 최상위 클래스가 CObject이기 때문이다. Single프로젝트에 생성된 CSingleDoc 클래스에는 DECLARE_SERIAL()과 IMPLEMENT_SERIAL() 매크로가 포함되지 않은 것을 확인할 수 있는데, 왜냐하면 직렬화에 필요한 기능이 이미

```
    DECLARE_DYNCREATE(CSingleDoc)
```

에 의해 구현되어 있기 때문이다. CFoo 클래스의 소스를 [예제 13.3]과 [예제

13.4]에 리스트하였다.

[예제 13.3] Foo.h

```
class CFoo
{
public:
    CFoo();
    virtual ~CFoo();

public:
    CString  m_strName;  // employee name
    int      m_nId;      // employee id

public:
    virtual void Serialize(CArchive& ar);
};//class CFoo
```

[예제 13.4] Foo.cpp

```
#include "stdafx.h"
#include "Single.h"
#include "Foo.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

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

CFoo::CFoo()
{

}

CFoo::~CFoo()
```

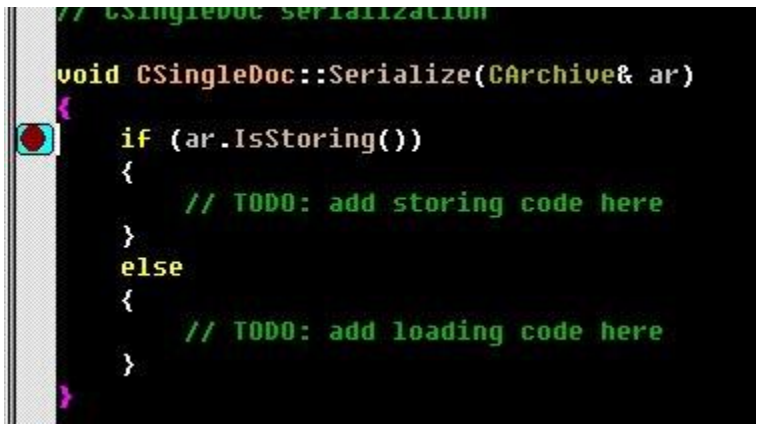
```
{

}


void CFoo::Serialize(CArchive& ar)
{
    if ( ar.IsStoring() )
    {
        AfxTrace( "Storing...\n" );
    }
    else
    {
        AfxTrace( "Loading...\n" );
    }//if.. else..
}//CFoo::Serialize()
```

MFC의 실제 동작

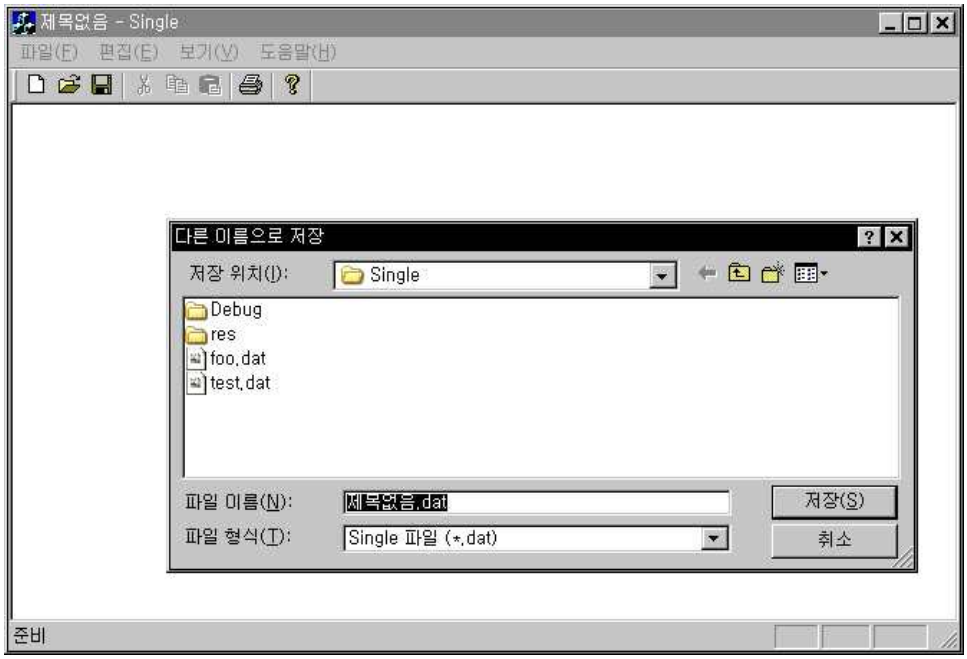
이제 MFC의 실제 동작을 확인해 보자. 먼저 [그림 13.6]처럼 CSingleDoc의 Serialize()함수에 브레이크 포인트(break point)를 설정한다.



[그림 13.6] 브레이크 포인트 설정: Serialize()의 첫 번째 줄에 F9를 눌러 브레이크 포인트를 설정한다.

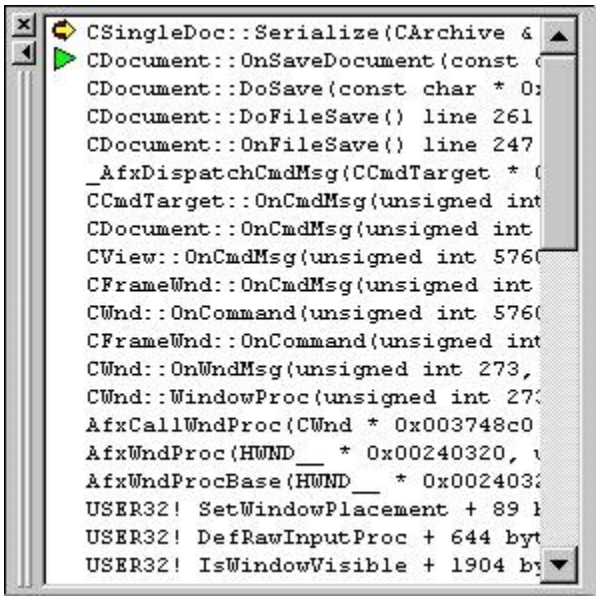
프로그램이 브레이크 포인트에서 멈추도록 하기 위해, 디버그 런(debug

run)한다.



[그림 13.7] 파일→다른 이름으로 저장(A)...의 선택: 파일 저장을 선택하여 Serialize()에 설정된 브레이크 포인트에서 멈추도록 한다.

프로그램이 실행되면 파일 메뉴에서 “다른 이름으로 저장(A)...”을 선택하여 대화상자에서 디폴트 파일이름을 선택하고 ”저장“ 버튼을 누른다.



[그림 13.8] CSingleDoc::Serialize()의 스택 상태:

그러면 설정된 브레이크 포인트에서 디버깅 모드가 시작되는데 Call Stack 윈도우를 확인해 보자. Serialize()를 호출한 상위 함수를 선택하면 CDocument::OnSaveDocument()의 소스를 확인해 볼 수 있다. 소스는 아래와 같다.

```
BOOL CDocument::OnSaveDocument(LPCTSTR lpszPathName)
{
    CFileException fe;
    CFile* pFile = NULL;
    pFile = GetFile(lpszPathName, CFile::modeCreate |
        CFile::modeReadWrite | CFile::shareExclusive, &fe); // (1)

    if (pFile == NULL)
    {
        ReportSaveLoadException(lpszPathName, &fe,
            TRUE, AFX_IDP_INVALID_FILENAME);
        return FALSE;
    }

    CArchive saveArchive(pFile, CArchive::store |\
 CArchive::bNoFlushOnDelete); // (2)
```

```
    saveArchive.m_pDocument = this;
    saveArchive.m_bForceFlat = FALSE;
    TRY
    {
        CWaitCursor wait;
        Serialize(saveArchive);     // save me // (3)
        saveArchive.Close();
        ReleaseFile(pFile, FALSE);
    }
    CATCH_ALL(e)
    {
        ReleaseFile(pFile, TRUE);

        TRY
        {
            ReportSaveLoadException(lpszPathName, e,
                TRUE, AFX_IDP_FAILED_TO_SAVE_DOC);
        }
        END_TRY
        DELETE_EXCEPTION(e);
        return FALSE;
    }
    END_CATCH_ALL

    SetModifiedFlag(FALSE);     // back to unmodified

    return TRUE;        // success
}
```

MFC의 OnSaveDocument()의 소스를 보면, (1) 열려진 파일 객체의 시작 주소를 얻고, (2) CArchive 객체를 만들고, (3) Serialize()를 호출하는 것을 확인할 수 있다.

직렬화 정리

직렬화에 관한 내용을 정리하면 다음과 같다. MFC는 객체의 로드와 저장을 위해 직렬화를 사용한다. AppWizard가 생성한 코드에서 직렬화를 구현하기 위해서는 다큐먼트 클래스의 Serialize()를 오버라이드한다.

또한, 제작하는 클래스에 직렬화 기능을 추가하기 위해서는 반드시 CObject를 상속 받고 헤더 파일에 DECLARE_SERIAL(), 구현 파일에

IMPLEMENT_SERIAL()을 추가한다.

직렬화 소스 분석

<절도비라>
이제 7장에서 처음 작성했던 MFC Single 프로젝트의 직렬화 관련 소스를 분석해보자. 또한 OnDraw()의 실제 동작을 살펴보고 다큐먼트,뷰와 메인 프레임의 메시지 맵 순서를 살펴본다.
</절도비라>

이제 7장에서 작성한 Single 프로젝트의 Document와 View에 관한 소스를 분석할 차례이다. 소스가 변경되었으므로 직렬화를 위해 설명한 프로젝트를 닫고, 다시 7장의 프로젝트를 열자.

대부분의 매크로는 이미 설명하였다. 이 장에서는 앞 장들에서 설명되지 않은 부분에 대해서만 설명할 것이다.

SingleDoc.h

아래에 CSingleDoc.h의 소스를 리스트하였다.

[예제 13.5] SingleDoc.h

```
001: // SingleDoc.h : interface of the CSingleDoc class
002: //
003: /////////////////////////////////////////////////////////////////////////
004:
005: #if !defined(AFX_SINGLEDOC_H__D322E434_F245_49FD_AF25_
__INCLUDED_)
006: #define AFX_SINGLEDOC_H__D322E434_F245_49FD_AF25_D5186685
__INCLUDED_
007:
008: #if _MSC_VER > 1000
009: #pragma once
010: #endif // _MSC_VER > 1000
011:
012:
```

```
013: class CSingleDoc : public CDocument
014: {
015: protected: // create from serialization only
016:     CSingleDoc();
017:     DECLARE_DYNCREATE(CSingleDoc)
018:
019: // Attributes
020: public:
021:
022: // Operations
023: public:
024:
025: // Overrides
026:     // ClassWizard generated virtual function overrides
027:     //{{AFX_VIRTUAL(CSingleDoc)
028:     public:
029:     virtual BOOL OnNewDocument();
030:     virtual void Serialize(CArchive& ar);
031:     //}}AFX_VIRTUAL
032:
033: // Implementation
034: public:
035:     virtual ~CSingleDoc();
036: #ifdef _DEBUG
037:     virtual void AssertValid() const;
038:     virtual void Dump(CDumpContext& dc) const;
039: #endif
040:
041: protected:
042:
043: // Generated message map functions
044: protected:
045:     //{{AFX_MSG(CSingleDoc)
046:         // NOTE - the ClassWizard will add and remove member
             //functions here.
047:         //    DO NOT EDIT what you see in these blocks of
             // generated code !
048:     //}}AFX_MSG
049:     DECLARE_MESSAGE_MAP()
050: };
051:
052: /////////////////////////////////////////////////////////////////////////
053:
```

```
054: //{{AFX_INSERT_LOCATION}}
055: // Microsoft Visual C++ will insert additional
     // declarations immediately before the previous line.
056:
057: #endif // !defined(AFX_SINGLEDOC_H__D322E434_F245_49FD_AF25`
_D518668521F3__INCLUDED_)
```

CSingleDoc 클래스는 CDocument를 상속 받아 구현한다. 29,30번 줄을 보자.

```
029:     virtual BOOL OnNewDocument();
030:     virtual void Serialize(CArchive& ar);
```

위 두 줄은 역시 특별한 주석 //{{AFX_VIRTUAL에 감싸여 있다. 이것은 이 함수가 클래스 위저드가 유지하는 가상 함수임을 의미한다. 다시 한번 상기하면, 클래스 위저드는 두 가지 종류의 함수에 대한 매핑(mapping)을 지원한다. 하나는 MFC의 구조 혹은 윈도우 메시지의 흐름으로 인해 필요한 상위 클래스에 이미 정의된 가상 함수에 대한 맵이고, 다른 하나는 WM_로 시작하는 실제 윈도우 메시지의 핸들러를 맵하는 것이다.

이 주석은 두 개의 함수가 가상 함수로 맵되는 것을 나타낸다. 또한 이것은 함수의 선언 변경자로 virtual이 붙은 이유이다.

OnNewDocument()는 문서를 새로 만들어야 할 때 불리는 함수이다. 문서를 새로 만드는 일이 항상 윈도우의 메시지와 대응되지 않음에 주목하라. 예를 들면, File->New를 선택하면, 문서를 새로 만들어야 한다. 이것이 메뉴 선택 메시지(WM_COMMAND)에 대응하는 핸들러가 OnNewDocument()임을 의미하지는 않는다. File->New 메시지에 대응하는 이벤트 핸들러는 따로 있다. OnNewDocument()는 메시지 핸들러가 불리기 전에, MFC의 구조상 불리는 함수이다.

또한 프로그램을 처음 시작할 때, 문서는 새로 만들어져야 할 것이다. 이 때에도 OnNewDocument()가 호출된다. 자 이렇게 윈도우 메시지에 대응되는 것이 아니고, 내부적인 제어의 흐름상 필요한 함수들은 가상 함수로 맵된다는 것을 다시 한번 상기하자.

MFC로 프로그램할 때 이러한 가상 함수들에 대해서는 다음과 같은 사실들을 기억하고 있어야 한다.

① 이 함수 전후에 다른 어떤 함수들이 불리는가?
② 이 함수는 언제 불리는가?

자 OnNewDocument()가 문서를 새로 만들어야 할 때 불린다는 것을 기억하자. 그러므로, 이미 CSingleDocument 객체는 만들어진 상태이므로, 문서의 초기화 작업 등을 이곳에서 한다.

```
030:    virtual void Serialize(CArchive& ar);
```

30번 줄에서 직렬화를 위한 Serialize() 함수의 선언을 발견할 수 있다. 이제 CSingleDoc의 구현 파일을 살펴보자.

SingleDoc.cpp

CSingleDoc 클래스의 구현 파일인 SingleDoc.cpp를 [예제 13.6]에 리스트하였다.

[예제 13.6] SingleDoc.cpp

```
001: // SingleDoc.cpp : implementation of the CSingleDoc class
002: //
003:
004: #include "stdafx.h"
005: #include "Single.h"
006:
007: #include "SingleDoc.h"
008:
009: #ifdef _DEBUG
010: #define new DEBUG_NEW
011: #undef THIS_FILE
012: static char THIS_FILE[] = __FILE__;
013: #endif
014:
0                     1                     5
////////////////////////////////////////////////
/////
016: // CSingleDoc
017:
```

```
018: IMPLEMENT_DYNCREATE(CSingleDoc, CDocument)
019:
020: BEGIN_MESSAGE_MAP(CSingleDoc, CDocument)
021:     //{{AFX_MSG_MAP(CSingleDoc)
022:         // NOTE - the ClassWizard will add and remove
             // mapping macros here.
023:         //    DO NOT EDIT what you see in these blocks of
             // generated code!
024:     //}}AFX_MSG_MAP
025: END_MESSAGE_MAP()
026:
027: /////////////////////////////////////////////////
028: // CSingleDoc construction/destruction
029:
030: CSingleDoc::CSingleDoc()
031: {
032:     // TODO: add one-time construction code here
033:
034: }
035:
036: CSingleDoc::~CSingleDoc()
037: {
038: }
039:
040: BOOL CSingleDoc::OnNewDocument()
041: {
042:     if (!CDocument::OnNewDocument())
043:         return FALSE;
044:
045:     // TODO: add reinitialization code here
046:     // (SDI documents will reuse this document)
047:
048:     return TRUE;
049: }
050:
051:
052:
053: /////////////////////////////////////////////////
054: // CSingleDoc serialization
055:
056: void CSingleDoc::Serialize(CArchive& ar)
057: {
058:     if (ar.IsStoring())
```

```
059:     {
060:         // TODO: add storing code here
061:     }
062:     else
063:     {
064:         // TODO: add loading code here
065:     }
066: }
067:
068: /////////////////////////////////////////////////
069: // CSingleDoc diagnostics
070:
071: #ifdef _DEBUG
072: void CSingleDoc::AssertValid() const
073: {
074:     CDocument::AssertValid();
075: }
076:
077: void CSingleDoc::Dump(CDumpContext& dc) const
078: {
079:     CDocument::Dump(dc);
080: }
081: #endif //_DEBUG
082:
083: /////////////////////////////////////////////////
084: // CSingleDoc commands
```

Serialize()함수를 살펴보자. 이 함수는 56~66번 줄에 정의되어 있다.

```
056: void CSingleDoc::Serialize(CArchive& ar)
057: {
058:     if (ar.IsStoring())
059:     {
060:         // TODO: add storing code here
061:     }
062:     else
063:     {
064:         // TODO: add loading code here
065:     }
066: }
```

Serialize()함수의 내부를 보면, CArchive의 멤버 함수 IsStoring()을 호출하고 있다.

```
058:     if (ar.IsStoring())
059:     {
060:         // TODO: add storing code here
061:     }
062:     else
063:     {
064:         // TODO: add loading code here
065:     }
```

IsStoring()은 객체를 저장(store)하고 있을 때 1을 리턴한다. 즉 위의 if문의 의미는 객체를 저장하는 일을 60번 줄에 명시하고, 객체를 읽는(load) 일을 64번 줄에 명시하라는 의미이다. 예를 들면(i와 s가 클래스의 정수 멤버, 스트링 멤버라고 가정하자),

```
058:     if (ar.IsStoring())
059:     {
060:         // TODO: add storing code here
             ar << i << s;
061:     }
062:     else
063:     {
064:         // TODO: add loading code here
             ar >> i >> s;
065:     }
```

는 멤버 변수 i,s를 저장하고, 로드한다. 주의할 점은 저장한 순서대로 반드시 로드되어야 한다는 것이다. 즉

```
ar >> s >> i;
```

라고 사용하면, 바른 값이 읽히지 않을 것이다.

순수 C++로 RTTI, MessageMap, CDC등을 모두 구현해 보았다. 이것은 확실히 MFC의 소스를 이해하는데 도움이 된다. 어떤가? 시도해 보지 않았지만, 독자들이 CArchive를 직접 구현해 보는 것이!

이제 분석할 마지막 한 개의 클래스를 남겨 놓았다.

SingleView.h

자동으로 생성된 SingleView.h를 아래에 리스트하였다.

[예제 13.7] SingleView.h

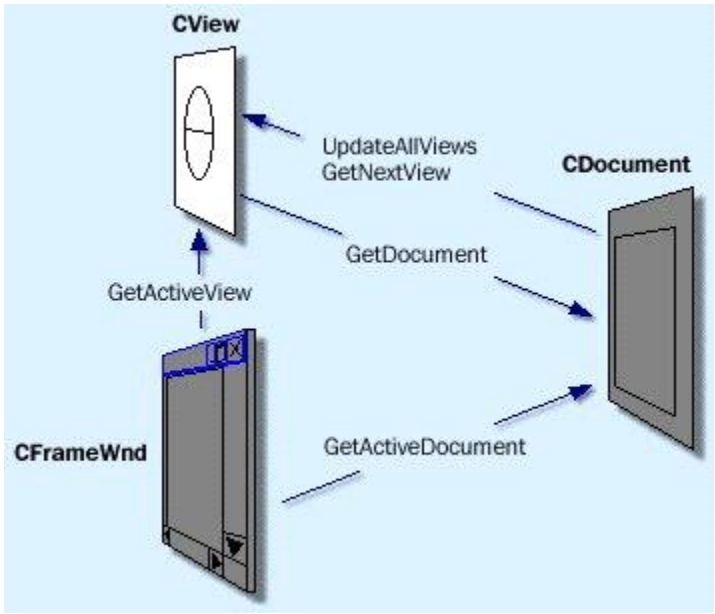
```
001: // SingleView.h : interface of the CSingleView class
002: //
003: /////////////////////////////////////////////////////////////////////////////
004:
005: #if !defined(AFX_SINGLEVIEW_H__5B7D9117_2B92_4840_A5D3
_F5741792729E__INCLUDED_)
006: #define AFX_SINGLEVIEW_H__5B7D9117_2B92_4840_A5D3\
_F5741792729E__INCLUDED_
007:
008: #if _MSC_VER > 1000
009: #pragma once
010: #endif // _MSC_VER > 1000
011:
012:
013: class CSingleView : public CView
014: {
015: protected: // create from serialization only
016:     CSingleView();
017:     DECLARE_DYNCREATE(CSingleView)
018:
019: // Attributes
020: public:
021:     CSingleDoc* GetDocument();
022:
023: // Operations
024: public:
025:
026: // Overrides
027:     // ClassWizard generated virtual function overrides
028:     //{{AFX_VIRTUAL(CSingleView)
029:     public:
030:       virtual void OnDraw(CDC* pDC);  // overridden to draw this
view
```

```
031:     virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
032:     protected:
033:     virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
034:     virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
035:     virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
036:     //}}AFX_VIRTUAL
037:
038: // Implementation
039: public:
040:     virtual ~CSingleView();
041: #ifdef _DEBUG
042:     virtual void AssertValid() const;
043:     virtual void Dump(CDumpContext& dc) const;
044: #endif
045:
046: protected:
047:
048: // Generated message map functions
049: protected:
050:     //{{AFX_MSG(CSingleView)
051:         // NOTE - the ClassWizard will add and remove
             // member functions here.
052:         //    DO NOT EDIT what you see in these blocks of
             // generated code !
053:     //}}AFX_MSG
054:     DECLARE_MESSAGE_MAP()
055: };
056:
057: #ifndef _DEBUG  // debug version in SingleView.cpp
058: inline CSingleDoc* CSingleView::GetDocument()
059:    { return (CSingleDoc*)m_pDocument; }
060: #endif
061:
062: /////////////////////////////////////////////////
063:
064: //{{AFX_INSERT_LOCATION}}
065: // Microsoft Visual C++ will insert additional
     // declarations immediately before the previous line.
066:
067: #endif // !defined(AFX_SINGLEVIEW_H__5B7D9117_2B92_4840_A5D3
_F5741792729E__INCLUDED_)
```

CSingleView는 CView를 상속받아 구현한다. 21번 줄에는 다큐먼트 객체의 주소를 얻는 멤버 함수가 선언되어 있다.

```
021:        CSingleDoc* GetDocument();
```

앞 장에서 다큐먼트/뷰 구조에 대해서 살펴 본 적이 있다. MFC는 메인프레임, 다큐먼트와 뷰의 상호 작용을 위해 아래 [그림 13.9]와 같은 함수를 제공한다. 반드시 암기하도록 하자.



[그림 13.9] 메인프레임, 다큐먼트와 뷰의 상호 작용을 위한 함수들

다큐먼트는 자신의 데이터가 바뀐 것을 뷰에 알려주기 위해, UpdateAllViews()를 멤버 함수로 가진다. CDocument에서 UpdateAllViews()를 호출하면, CView의 OnUpdate()가 호출된다. 또한 다중 뷰(multiple view) 응용 프로그램에서 다음 뷰(next view) 객체의 시작 주소를 얻기 위해 GetNextView()를 호출할 수 있다.

CView에서 CDocument 객체의 시작 주소를 얻기 위해 GetDocument()를 사용한다.

CDocument와 CView의 소유자인 CMainFrame은 현재의 활성화된 뷰와 다큐먼트를 얻기 위해 각각 GetActiveView()와 GetActiveDocument()를 호출한

다. 아래에 리스트한 함수를 기능과 더불어 반드시 암기하도록 하자.

① CView::GetDocument()
② CDocument::UpdateAllViews()
③ CView::OnUpdate()
④ CDocument::GetNextView()
⑤ CMainFrame::GetActiveView()
⑥ CMainFrame::GetActiveDocument()

위의 함수와 더불어 빈번히 사용되는 아래의 함수들도 암기하도록 하자.

⑦ CDC::GetSafeHdc() : HDC를 얻는다.
⑧ CWnd::GetSafeHwnd() : HWND를 얻는다.
⑨ ::AfxGetInstanceHandle() : HINSTANCE를 얻는다.

<여기서 잠깐>
MSDN 도움말을 참고로 하여, Afx...()로 시작하는 함수들을 정리하도록 한다. Afx...()로 시작하는 함수들은 MFC에서 싱글턴 패턴으로 구현된 전역 객체와 관계된 함수들이다.
</여기서 잠깐>

⑩ ::AfxGetMainWnd() : MainFrame윈도우 객체의 포인터를 얻는다.
⑪ ::AfxGetResourceHandle() : 리소스 핸들(HINSTANCE)을 얻는다.
⑫ ::AfxMessageBox() : 메시지 박스를 실행한다.

GetDocument()는 단순히 이미 초기화된 CView의 멤버인 m_pDocument를 리턴한다. 디버그 타겟인 경우 m_pDocument가 유효한 클래스인지 검사하는 추가적인 작업을 한다. 하지만, 릴리즈 타겟인 경우 이러한 작업은 이제 불필요할 것이다. 57번 줄을 보자.

```
057: #ifndef _DEBUG  // debug version in FirstView.cpp
058: inline CFirstDoc* CFirstView::GetDocument()
059:    { return (CFirstDoc*)m_pDocument; }
060: #endif
```

GetDocument()의 릴리즈 버전은 인라인 함수로 정의된다. 이제 30~33번 줄을 보자.

```
028:     //{{AFX_VIRTUAL(CFirstView)
```

```
029:     public:
030:     virtual void OnDraw(CDC* pDC);
    // overridden to draw this view
031:     virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
032:     protected:
033:     virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
```

클래스 위저드가 관리하는 코드로 몇 개의 가상함수가 정의되어 있다. OnDraw()는 윈도우의 클라이언트 영역이 무효화(invalidate)되었을 때, 즉 WM_PAINT 메시지가 발생했을 때 호출된다.

실제의 OnDraw()

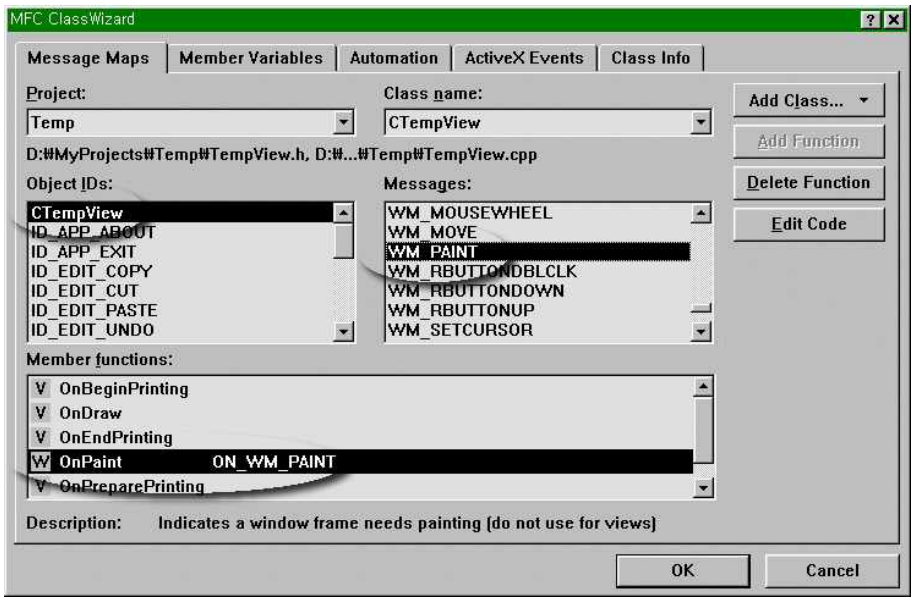
WM_PAINT 메시지가 발생했을 때, OnDraw()가 호출되는 것은 사실이지만, 항상 그런 것은 아니다. 실제로 WM_PAINT에 대응하는 메시지 핸들러는 OnPaint()로 가상 함수가 아니다. OnPaint()의 내부에서 OnDraw()를 호출하는 것이다.

OnDraw()가 항상 화면에 뭔가를 그릴 것이라고 기대해서는 안 된다. OnDraw()는 파라미터로 전달되는 디바이스 컨텍스트의 표면에 그리기 작업을 하는 것이다. 만약 파라미터로 전달되는 pDC가 프린터의 디바이스 컨텍스트라면 OnDraw()는 프린터에 출력을 하게 된다.

이것이 OnPaint()와 OnDraw()를 구분한 이유이다. MFC는 프로그래머로 하여금 그리는 작업을 OnDraw()에 집중하여 넣을 것을 강요한다. WM_PAINT 메시지가 발생했을 때 OnPaint()가 호출된다. OnPaint()는 화면에 대한 디바이트 컨텍스트 객체를 만들어 OnDraw()를 호출하며 파라미터로 디바이스 컨텍스트 객체의 포인터를 전달한다. 그래서 OnDraw()에서는 더 이상 디바이스 컨텍스트 객체를 만들 필요가 없다.

만약 프린팅 작업을 해야 한다면, 프린팅 작업의 내부에서 프린터에 대한 디바이스 컨텍스트를 객체를 만들어 OnDraw()의 파라미터로 전달할 것이다.

WM_PAINT를 직접 맵할 일을 거의 없지만, 이것이 불가능한 것은 아니다. 클래스 위저드를 이용하여 WM_PAINT를 맵해 보자.



[그림 13.10] WM_PAINT의 맵: 클래스 위저드를 사용하여 View 클래스에 WM_PAINT를 맵하면, OnPaint()의 골격이 생성된다.

클래스 위저드의 Messages 리스트 박스에서 WM_PAINT를 더블 클릭하면 OnPaint()의 골격이 생성된다. 이미 살펴본 대로 세 부분에 코드가 추가된다.

```
void CSingleView::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // TODO: Add your message handler code here

    // Do not call CView::OnPaint() for painting messages
}
```

OnPaint()의 골격을 보면 CPaintDC 객체를 만들고 있다. CPaintDC의 생성자에서 BeginPaint()를 호출하고, 파괴자에서 EndPaint()를 호출한다는 것을 상기하자. 주의하라. 이제 WM_PAINT가 발생했을 때, OnPaint()가 호출되고 OnDraw()는 호출되지 않는다. 만약 OnDraw()가 호출되게 하려면, OnPaint()에 아래처럼 소스를 추가해야 한다.

```
void CSingleView::OnPaint()
```

```
{
    CPaintDC dc(this); // device context for painting

    // TODO: Add your message handler code here
    OnDraw(&dc);
    // Do not call CView::OnPaint() for painting messages
}
```

이미 만들어진 OnPaint()에서는 위의 코드를 포함하고 있으므로, OnDraw()가 불린 것이다. 그래서 OnDraw()는 가상 함수로, OnPaint()는 메시지 맵으로 관리된다.

33번 줄부터 정의된 세 개의 가상 함수는 프린팅 작업과 연관된 함수이다. 각각의 함수는 프린팅 작업을 준비할 때, 프린팅 작업을 시작할 때, 프린팅 작업을 끝낼 때 불린다.

메시지가 맵되는 순서

윈도우 메시지를 맵하는 것이 CView, CDocument 및 CMainFrame에 모두 가능하다는 사실을 주의하자. 예를 들면, WM_LBUTTONDOWN 메시지는 CView와 CMainFrame에 맵 하는 것이 가능하다■. 두 곳 모두에 메시지 핸들

<여기서 잠깐>

CDocument에 WM_LBUTTONDOWN 메시지가 발생할 수 없다. CDocument 클래스는 윈도우를 가지지 않으므로, CDocument에 맵할 수 있는 유일한 메시지는 메뉴 선택에 대응한 WM_COMMAND 메시지 뿐이다.

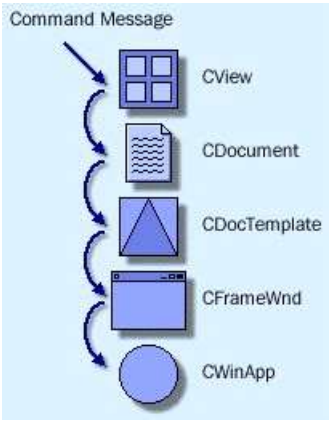
</여기서 잠깐>

러가 맵 되어 있다면, 어느 것이 호출될까? 매핑된 함수는 다음과 같은 순서로 검사된다.

① CView
② CDocument
③ CMainFrame
④ CWinApp

그러므로, WM_LBUTTONDOWN에 해당하는 OnLButtonDown()은 CView

의 것이 불릴 것이다(CView이외의 OnLButtonDown()은 불리지 않는다). 아래의 [그림 13.11]을 꼭 기억하도록 하자.



[그림 13.11] 윈도우 메시지가 MFC의 클래스에 도착하는 순서: View, Document와 MainFrame의 순으로 메시지의 맵 상태를 검사한다.

왜 MFC의 설계자는 이러한 방식의 맵을 설계한 것일까? 그것은 윈도우 메시지가 특정 클래스와 연관된 것이 아니기 때문이다. 예를 들면, WM_LBUTTONDOWN 메시지는 CMainFrame과 연관된 것인가? 아니면 CView와 연관된 것인가? 이것은 전적으로 응용 프로그램을 구현하는 사람에게 의존적인 것이다. 그래서 메시지를 맵하는 것이 모든 클래스에 가능하도록 허락되는 것이다.

두 곳 이상에 같은 메시지가 맵 되었을 때, 순서대로 검사된 처음의 메시지 핸들러만이 호출된다는 사실을 주의하자. 예를 들면 CView와 CMainFrame에 모두 OnLButtonDown()이 맵된 경우 CView의 OnLButtonDown()만이 호출된다.

만약 필요에 의해 두 곳 모두의 OnLButtonDown()이 호출되어야 한다면, 어떻게 할 것인가? 그것은 처음 호출될 OnLButtonDown()에 다른 윈도우의 멤버 함수 OnLButtonDown()을 호출하는 코드를 삽입함으로써 가능하다.

예를 들면, CView::OnLButtonDown()에서 CMainFrame의 OnLButtonDown()을 호출하기 위해 코드를 다음과 같이 작성할 수 있다.

```
void CSingleView::OnLButtonDown(UINT nFlags, CPoint point)
{
```

```
    // TODO: Add your message handler code here and/or call default
    AfxMessageBox("view");
    ((CMainFrame*)AfxGetMainWnd())->OnLButtonDown                    ;

    CView::OnLButtonDown(nFlags, point);
}
```

그리고 CMainFrame 등의 메시지 핸들러는 보호 멤버(protected member)이므로, CMainFrame의 클래스 선언 부분에서 protected:를 public:으로 다음과 같이 수정해야 한다(AfxGetMainWnd()가 리턴하는 값을 왜 불안전하게 - 서브타입의 원리를 어기면서 - CMainFrame*로 캐스팅해야하는지에 대해서 독자들 스스로 답해보기 바란다).

```
...
public:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

자동으로 생성된 소스를 수정하면서까지 이렇게 구현하는 것이 바람직한 것일까? 그렇지 않다. 만약 독자들이 위의 예에서처럼 매핑 된 두 곳 이상의 함수가 호출되도록 코드를 설계했다면 설계에 문제가 있는 것이라고 말할 수 있다. 자동으로 생성된 소스를 수정하고 싶지 않다면 SendMessage()나 PostMessage()를 이용할 수도 있다.

SingleView.cpp

자동으로 생성된 SingleView.cpp의 소스를 아래에 리스트하였다.

[예제 13.8] SingleView.cpp

```
001: // SingleView.cpp : implementation of the CSingleView class
002: //
003:
004: #include "stdafx.h"
```

```
005: #include "Single.h"
006:
007: #include "SingleDoc.h"
008: #include "SingleView.h"
009:
010: #ifdef _DEBUG
011: #define new DEBUG_NEW
012: #undef THIS_FILE
013: static char THIS_FILE[] = __FILE__;
014: #endif
015:
016: /////////////////////////////////////////////////
017: // CSingleView
018:
019: IMPLEMENT_DYNCREATE(CSingleView, CView)
020:
021: BEGIN_MESSAGE_MAP(CSingleView, CView)
022:     //{{AFX_MSG_MAP(CSingleView)
023:         // NOTE - the ClassWizard will add and remove
             // mapping macros here.
024:         //    DO NOT EDIT what you see in these blocks of
             // generated code!
025:     //}}AFX_MSG_MAP
026:     // Standard printing commands
027:     ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
028:     ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
029:     ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
030: END_MESSAGE_MAP()
031:
032: /////////////////////////////////////////////////
033: // CSingleView construction/destruction
034:
035: CSingleView::CSingleView()
036: {
037:     // TODO: add construction code here
038:
039: }
040:
041: CSingleView::~CSingleView()
042: {
043: }
044:
045: BOOL CSingleView::PreCreateWindow(CREATESTRUCT& cs)
```

```
046: {
047:     // TODO: Modify the Window class or styles here by modifying
048:     //  the CREATESTRUCT cs
049:
050:     return CView::PreCreateWindow(cs);
051: }
052:
053: /////////////////////////////////////////////////
054: // CSingleView drawing
055:
056: void CSingleView::OnDraw(CDC* pDC)
057: {
058:     CSingleDoc* pDoc = GetDocument();
059:     ASSERT_VALID(pDoc);
060:     // TODO: add draw code for native data here
061: }
062:
063: /////////////////////////////////////////////////
064: // CSingleView printing
065:
066: BOOL CSingleView::OnPreparePrinting(CPrintInfo* pInfo)
067: {
068:     // default preparation
069:     return DoPreparePrinting(pInfo);
070: }
071:
072: void CSingleView::OnBeginPrinting(CDC* /*pDC*/,
        CPrintInfo* /*pInfo*/)
073: {
074:     // TODO: add extra initialization before printing
075: }
076:
077: void CSingleView::OnEndPrinting(CDC* /*pDC*/,
        CPrintInfo* /*pInfo*/)
078: {
079:     // TODO: add cleanup after printing
080: }
081:
082: /////////////////////////////////////////////////
083: // CSingleView diagnostics
084:
085: #ifdef _DEBUG
086: void CSingleView::AssertValid() const
```

```
087: {
088:     CView::AssertValid();
089: }
090:
091: void CSingleView::Dump(CDumpContext& dc) const
092: {
093:     CView::Dump(dc);
094: }
095:
096: CSingleDoc* CSingleView::GetDocument() // non-debug version
                                            // is inline
097: {
098:     ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CSingl
099:     return (CSingleDoc*)m_pDocument;
100: }
101: #endif //_DEBUG
102:
103: /////////////////////////////////////////////////
104: // CSingleView message handlers
```

먼저 27번 줄의 매크로를 보자.

027: **ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)**

ON_COMMAND() 매크로는 WM_COMMAND 메시지에 해당하는 매크로이다. WM_COMMAND 메시지는 메뉴 선택 혹은 차일드 윈도우 컨트롤이 부모에게 메시지를 보낼 때 발생한다는 사실을 상기하자.

045: **BOOL CFirstView::PreCreateWindow(CREATESTRUCT& cs)**

45번 줄의 PreCreateWindow()는 CView윈도우를 만들기 전에 불린다는 것을 주의하자. 즉, 이미 살펴본 CMainFrame의 PreCreateWindow()는 MainFrame 윈도우를 만들기 전에, CView의 PreCreateWindow()는 클라이언트 영역을 차지하는 뷰 윈도우를 만들기 전에 호출된다.

056: **void CFirstView::OnDraw(CDC* pDC)**

OnDraw()가 받는 파라미터는 CDC* 타입이다. 이미 OnPaint()에서 CPaintDC 의 객체가 만들어진 상태이므로, 포인터를 받는 것은 타당하다. 또

한 CDC는 CPaintDC의 상위 클래스이므로 서브 타입의 원리(subtype principle)를 위배하지 않는다.

요약

이 장에서 직렬화의 원리를 살펴보았고, 남아있던 Single 프로젝트의 뷰와 다큐먼트의 소스를 모두 분석했다.

- **직렬화**란 분산된 데이터들을 "차례대로" "어느 곳"에 저장하는 것을 말하며, 일반적으로 "어느 곳"이란 파일이나, 네트웍 스트림 등이다.
- MFC의 직렬화는 파일을 직접 접근하지 않고 **CArchive**라는 브리지(bridge)를 사용한다.
- **DECLARE_SERIAL()**은 직렬화를 지원해야 하는 클래스가 포함해야할 매크로이다.
- **CArchive::IsStoring()**은 객체를 저장하고 있는 경우 TRUE를 리턴한다.
- 직렬화를 하는 경우 **Serialize()**가 호출된다.
- CView와 CDocument에 같은 윈도우 메시지가 맵되어 있는 경우, CDocument의 핸들러는 호출되지 않는다.
- WM_COMMAND 메시지를 표현하는 메시지 맵의 엔트리는 **ON_COMMAND()**이다.
- 7장에서 처음 작성한 Single MFC 프로젝트의 소스 분석을 모두 마쳤다!

이제 시작이다!

이제 코드 분석을 모두 마쳤다. 그렇다면 이제 끝인가? 그렇지 않다. 이제 겨우 자동으로 생성된 코드의 분석을 마쳤을 뿐이다.

많은 MFC 책들은 MFC의 각종 기교를 설명하고 있다. 이 책은 그러한 기교를 설명하지 않았다. 다른 적절한 윈도우즈 프로그래밍 교재를 선택하여 이제 프로그래밍을 본격적으로 시작할 때이다. 윈도우즈 프로그래밍과 관련해서 공부해야 할 큰 주제들에는 다음과 같은 것들이 있다.

- COM, DCOM, COM+
- GDI+
- ATL, WTL
- Shell Programming
- Network Programming, DB Programming

놀랍게도 하나하나의 공부를 마쳐가면, 공부해야 할 것들은 오히려 하나하나씩 늘어갈 것이다. 우리는 평생 공부해야 한다. 프로그래밍을 공부해야 하고, 사랑하는 방법을 공부해야 한다. 독자들의 시작을 소망한다. 성공을 소망하지는 않는다. 시작하면 성공할 것이므로!

[문서의 끝]