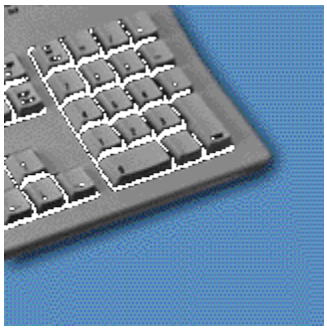


11. CMainFrame의 소스 분석

<장도비라>
이 장에서는 7장에서 작성했던 MFC Single 프로젝트에서 MainFrame의 소스를 분석한다. 또한 MFC가 다양한 타입의 윈도우 메시지를 일관되게 관리하는 방법을 살펴본다. Single 프로젝트의 다이얼로그에 관한 분석은 12장에서, 다큐먼트와 뷰에 관한 마지막 소스 분석은 13장에서 할 것이다.
</장도비라>

이 장에서는 7장에서 작성한 Single 프로젝트에서 CMainFrame 클래스의 소스를 분석할 것이다. 이 장에서 살펴볼 내용은 다음과 같다.

- Single 프로젝트의 CMainFrame 소스 분석
- MFC가 다양한 타입의 윈도우 메시지를 일관되게 관리하는 방법

CMainFrame의 소스 분석

<절도비라>
이 절에서는 7장에서 작성했던 MFC Single 프로젝트에서 MainFrame의 소스를 분석한다. CMainFrame의 계층 구조를 살펴보고, MFC에서 정의된 DECLARE_DYNCREATE 매크로 등을 찾고 이해한다. 또한, 가상 함수와 일반적인 메시지 맵으로 구분하여 관리되는 윈도우즈 메시지를 이해하고, 서로 다르게 해석해야 하는 메시지들을 시그너처(signature)로 구분하는 방법을 살펴본다.
</절도비라>

이제 7장에서 작성했던 Single 프로젝트를 다시 열고 메인 프레임의 소스를 다시 살펴보자. 이 장에서는 메인 프레임 클래스의 RTTI 부분을 중심적으로 살펴보고, 나머지 뷰와 다큐먼트의 소스는 13장에서 살펴 볼 것이다.

MainFrm.h

자동으로 생성된 CMainFrame 클래스의 헤더 파일을 [예제 11.1]에 리스트하였다(줄 앞의 세자리 숫자는 설명을 위해 붙인 것으로 소스에는 포함되어 있지 않다).

[예제 11.1] MainFrm.h

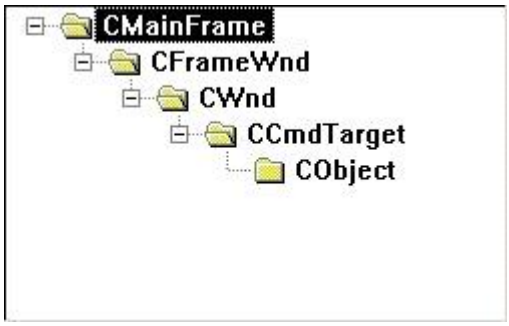
```
001: // MainFrm.h : interface of the CMainFrame class
002: //
003: /////////////////////////////////////////////////////////////////////////
004:
005: #if !defined (AFX_MAINFRM_H__236FD24B_AD9C_11D2_BB41_008048E104
__INCLUDED_)
006: #define AFX_MAINFRM_H__236FD24B_AD9C_11D2_BB41_008048E104
__INCLUDED_
007:
008: #if _MSC_VER > 1000
009: #pragma once
010: #endif // _MSC_VER > 1000
011:
012: class CMainFrame : public CFrameWnd
013: {
014:
015: protected: // create from serialization only
016: CMainFrame();
017:     DECLARE_DYNCREATE(CMainFrame)
018:
019: // Attributes
020: public:
021:
022: // Operations
023: public:
024:
025: // Overrides
026:     // ClassWizard generated virtual function overrides
027:     //{{AFX_VIRTUAL(CMainFrame)
028:     virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
029:     //}}AFX_VIRTUAL
```

```
030:
031: // Implementation
032: public:
033:     virtual ~CMainFrame();
034: #ifdef _DEBUG
035:     virtual void AssertValid() const;
036:     virtual void Dump(CDumpContext& dc) const;
037: #endif
038:
039: protected:  // control bar embedded members
040:     CStatusBar  m_wndStatusBar;
041:     CToolBar    m_wndToolBar;
042:
043: // Generated message map functions
044: protected:
045:     //{{AFX_MSG(CMainFrame)
046:     afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
047:         // NOTE - the ClassWizard will add and remove member
            //functions here.
048:         //    DO NOT EDIT what you see in these blocks of
            // generated code!
049:     //}}AFX_MSG
050:     DECLARE_MESSAGE_MAP()
051: };
052:
053: /////////////////////////////////////////////////////////////////////////////
054:
055: //{{AFX_INSERT_LOCATION}}
056: // Microsoft Visual C++ will insert additional declarations
     // immediately before the previous line.
057:
058: #endif // !defined\
(AFX_MAINFRM_H__236FD24B_AD9C_11D2_BB41_008048E104
```

MainFrame 클래스는 타이틀 바, 메뉴 바, 툴 바 및 상태 바를 포함하는 윈도우를 나타내는 것을 상기하자. 비록 View 클래스의 윈도우가 MainFrame 클래스의 사용자 영역(client area)에 나타나는 것은 사실이지만, MainFrame과 View클래스의 관계는 HAVE-A 관계가 아니다. 하지만, MainFrame 클래스는 상태 바와 툴 바를 분명 포함해야(have-a relationship) 할 것이다.

```
012: class CMainFrame : public CFrameWnd
```

12번 줄은 CMainFrame을 만들기 위해, CFrameWnd로부터 상속받는 것을 나타낸다. CMainFrame에서 Alt+F12를 눌러 클래스 계층을 확인할 수 있다.



[그림 11.1] CFrameWnd의 계보: 윈도우 메시지를 받으므로, CCmdTarget에서 파생된다. 또한 윈도우를 가지므로, CWnd에서 파생된다.

```
CCmdTarget <- CWnd <- CFrameWnd
```

CFrameWnd는 윈도우 메시지(CCmdTarget)를 받아야하며, 윈도우(CWnd)를 가지므로, 위와 같은 구조를 가진다.

```
017:     DECLARE_DYNCREATE(CMainFrame)
```

17번 줄은 동적으로 자신(CMainFrame)을 생성하기 위해 필요한 매크로이다. F12를 눌러 매크로의 정의를 AFX.H의 772번 줄에서 찾을 수 있다. MFC 소스는 아래와 같다.

```
#define DECLARE_DYNCREATE(class_name) \
        DECLARE_DYNAMIC(class_name) \
        static CObject* PASCAL CreateObject();
```

매크로는 DECLARE_DYNAMIC 매크로를 포함하고, 자신을 동적으로 생성하는 정적 멤버 함수의 선언이 있다.

DECLARE_DYNAMIC 매크로는 같은 파일 759번 줄에 있다.

```
#define DECLARE_DYNAMIC(class_name) \
public: \
```

```
    static const AFX_DATA CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const; \
```

실행시 CRuntimeClass의 주소를 얻는 멤버 함수인 GetRuntimeClass()함수가 매크로에 포함된 사실을 제외하곤, 우리가 작성한 소스와 같다. GetRuntimeClass()는 12장에서 DDX를 작성하면서 살펴 볼 것이다.

27~29번 줄에는 클래스 위저드가 생성하고 유지하는 코드 부분이 발견된다.

```
026:    // ClassWizard generated virtual function overrides
027:    //{{AFX_VIRTUAL(CMainFrame)
028:    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
029:    //}}AFX_VIRTUAL
```

클래스 헤더에서 클래스 위저드가 유지하는 코드 부분은 45~49번 줄에서도 발견할 수 있다.

```
045:    //{{AFX_MSG(CMainFrame)
046:    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
047:        // NOTE - the ClassWizard will add and remove member
            //functions here.
048:        //    DO NOT EDIT what you see in these blocks of
            //generated code!
049:    //}}AFX_MSG
```

27번과 29번 줄 사이는 가상함수가 선언되는 곳이며, 45번과 49번 줄은 메시지 맵의 함수가 선언되는 곳이다. 우리는 9장에서 MFC의 가상 함수를 살펴본 적이 있다. 다시 가상 함수와 메시지 핸들러의 차이점을 정리해 보자.

가상 함수 vs. 메시지 핸들러

가상함수는 public 멤버이며, 메시지 핸들러는 protected 멤버이다. 아래에 둘 간의 차이점을 정리한 표가 있다.

윈도우 구조상 필요한 함수	윈도우 메시지에 대응하는 함수
1. 가상함수 2. 접근 권한은 public 3. 윈도우 메시지의 순서 혹은 MFC의 내부 구조상 필요한 함수	1. 메시지 핸들러 2. 접근 권한은 protected 3. WM_...로 시작하는 윈도우 메시지에 대응하는 이벤트 핸들러

표 11.1. 메시지 맵이 가상 함수와 이벤트 핸들러(윈도우 메시지에 대응하는)를 구분하는 이유: 가상 함수로 매핑 되는 이벤트 핸들러는 실제 윈도우 메시지에 대응하는 것이 아니다.

가장 큰 차이점은 가상 함수는 MFC의 설계 구조상 필요한 함수이고, 메시지 핸들러는 윈도우 메시지에 대응하는 함수라는 것이다. 물론 둘간은 긴밀하게 연관되어 있다.

WM_CREATE 메시지를 고려해 보자. 이 메시지는 윈도우가 만들어 질 때, 엄격히 말하면, 윈도우가 화면에 나타나기 전, 윈도우를 그리고 유지하기 위한 모든 정보가 메모리에 할당되었을 때 발생하는 메시지이다.

사용자가 설치한 윈도우 프로시저에서 이 메시지를 첫 번째 받게 되는 이유는, InitInstance()에서 윈도우 클래스를 등록하고, CreateWindow()를 호출하기 때문이다. CreateWindow() 호출 직후, 윈도우 프로시저는 첫 번째 WM_CREATE 메시지를 받게 된다. 우리가 순수한 C++로 작성했던 소스 부분을 다시 고려해 보자.

```
...
wndclass.cbSize         =sizeof(wndclass);
wndclass.style          =CS_HREDRAW|CS_VREDRAW;
wndclass.lpfnWndProc    =WndProc;
wndclass.cbClsExtra     =0;
wndclass.cbWndExtra     =0;
wndclass.hInstance      =hInstance;
wndclass.hIcon          =LoadIcon(NULL,IDI_APPLICATION);
wndclass.hCursor        =LoadCursor(NULL,IDC_ARROW);
wndclass.hbrBackground  =(HBRUSH)GetStockObject(WHITE_BRUSH);
wndclass.lpszMenuName   =NULL;
wndclass.lpszClassName  =szAppName;
wndclass.hIconSm        =LoadIcon(NULL,IDI_APPLICATION);

RegisterClassEx(&wndclass);

hwnd=CreateWindow(
    szAppName,                  //window class name
```

```
        "The Hello Program",        //window caption
        WS_OVERLAPPEDWINDOW,        //window style
        CW_USEDEFAULT,              //initial x position
        CW_USEDEFAULT,              //initial y position
        CW_USEDEFAULT,              //initial x size
        CW_USEDEFAULT,              //initial y size
        NULL,                       //parent window handle
        NULL,                       //window menu handle
        hInstance,                  //program instance handle
        NULL);                      //creation parameters

    ShowWindow(hwnd,iCmdShow);
    UpdateWindow(hwnd);
    ...
```

위의 소스에서 보듯이 RegisterClassEx() 이후, CreateWindow()를 호출했다. WM_CREATE에 대응하는 OnCreate()는 이미 이벤트 핸들러가 맵된 상태이다. 왜냐하면, WM_CREATE는 반드시 발생하는(응용 프로그램이 윈도우를 가진다면) 메시지이기 때문이다. 만약, CreateWindow() 호출 이전, 윈도우의 모양을 바꾸는 등의 필요한 작업이 있다면, RegisterClassEx()와 CreateWindow() 함수 사이에 다음과 같은 함수를 만들 수 있다.

```
PreCreateWindow();
```

그러면 C++ 소스의 모양은 아래처럼 될 것이다.

```
    ...
    CREATESTRUCT cs;
    ...
    wndclass.hIconSm        =LoadIcon(NULL,IDI_APPLICATION);

    RegisterClassEx(&wndclass);
    PreCreateWindow(cs);
    hwnd=CreateWindow(
        szAppName,                  //window class name
        "The Hello Program",        //window caption
        ...,&cs);
    ...
```

CreateWindow()의 마지막 파라미터가 NULL에서 &cs로 바뀌었음에 주목

하라. 이것은 CREATESTRUCT라고 불리는 구조체의 시작 주소이다. 윈도우가 만들어질 때, 이 구조체의 필드를 참고로하여 윈도우의 모양(style)을 결정하는 것이다. 그리고, 여러분이 만든 PreCreateWindow()에서는 이 필드를 적당하게 설정하는 루틴이 포함되어야 할 것이다. CREATESTRUCT 구조체는 아래와 같이 정의되어 있다.

```
typedef struct tagCREATESTRUCT { // cs
    LPVOID    lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCTSTR   lpszName;
    LPCTSTR   lpszClass;
    DWORD     dwExStyle;
} CREATESTRUCT;
```

MFC에도 예상대로 PreCreateWindow()가 이미 만들어져 있다. 물론 사용자가 이 함수를 클래스 위저드로 매핑하여 기능을 추가할 수 있다. 여기에 해결의 열쇠가 있다. PreCreateWindow()는 비록 클래스 위저드가 매핑을 지원하는 이벤트 핸들러이기는 하지만(이것은 분명 윈도우가 만들어지기 전의 이벤트에 대응한다), WM_으로 시작하는 윈도우 메시지에 대응하는 것이 아니다.

이렇게 윈도우 메시지에 대응하는 것이 아니라, 윈도우의 구조, 혹은 MFC의 다큐먼트/뷰 아키텍처의 구조상 불리는 함수는 이미, 상위 클래스에 만들어 두었다. 또한 이러한 함수들은 상위 클래스 객체의 포인터를 사용하여, 하위 클래스 객체의 오버라이드된 멤버 함수를 호출할 수 있어야하므로(subtype의 원리), 가상함수로 선언하는 것이 마땅하다.

그래서 클래스 위저드는 가상 함수와 콜백 함수에 대한 두 가지 형식의 매핑을 지원하는 것이다. 독자들은, 후에 가상 함수로 존재하는 중요한 몇 가지 이벤트 핸들러에 대해 배우게 될 것이다. 또한 On...()으로 시작하는 함수가 모두 윈도우 메시지에 대응한다고 생각하지 말라. 몇 가상함수 들은 윈도우 메시지에 대응하지 않음에도 불구하고, 함수의 이름은 On...()으로 시작한다.

아래의 문제는 독자들 스스로 답해보기 바란다.

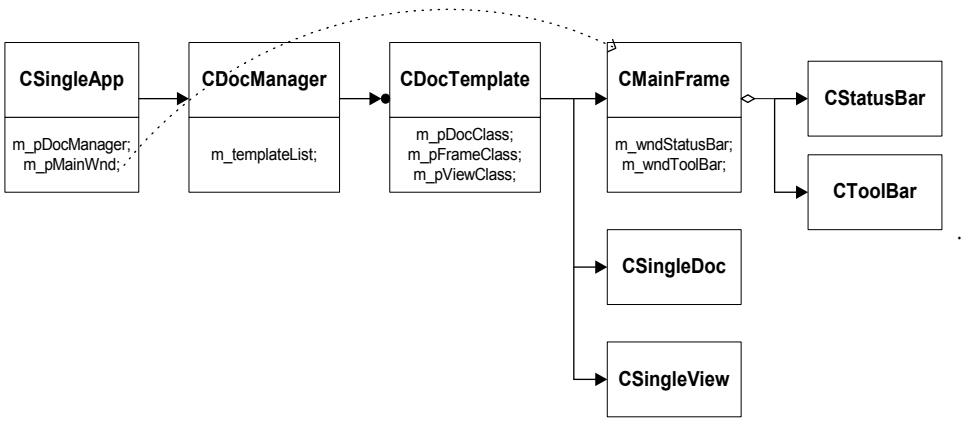
"왜 가상 함수는 접근 권한이 public 이고, 이벤트 핸들러는 protected 인가?"

```
034: #ifdef _DEBUG
035:     virtual void AssertValid() const;
036:     virtual void Dump(CDumpContext& dc) const;
037: #endif
```

34~37번 줄은 빌드 타겟이 디버그 모드일 때만 사용하는 함수들의 선언이다. 이러한 함수들은 릴리즈(release) 모드 일 때 분명히 제거되어야 한다.

```
040:     CStatusBar  m_wndStatusBar;
041:     CToolBar    m_wndToolBar;
```

40~41번 줄을 보면 MainFrame 객체가 상태바(CStatusBar)와 툴바(CToolBar) 객체를 포함(have-a)하는 것을 알 수 있다. 아래의 [그림 11.2]는 CWinApp, CMainFrame, CDocument, CView, CStatusBar 및 CToolBar의 관계를 나타낸다.



[그림 11.2] MFC 응용 프로그램의 클래스 포함 관계

```
050:     DECLARE_MESSAGE_MAP()
```

50번 줄에서는 메시지 맵을 선언하는 매크로를 볼 수 있다. 이제 메인 프레

임의 구현 파일을 살펴보자.

MainFrm.cpp

CMainFrame의 구현 파일(MainFrm.cpp)를 아래 [예제 11.2]에 리스트하였다.

[예제 11.2] MainFrm.cpp

```
001: // MainFrm.cpp : implementation of the CMainFrame class
002: //
003:
004: #include "stdafx.h"
005: #include "First.h"
006:
007: #include "MainFrm.h"
008:
009: #ifdef _DEBUG
010: #define new DEBUG_NEW
011: #undef THIS_FILE
012: static char THIS_FILE[] = __FILE__;
013: #endif
014:
015: /////////////////////////////////////////////////////////////////////////
016: // CMainFrame
017:
018: IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
019:
020: BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
021:     //{{AFX_MSG_MAP(CMainFrame)
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:     ON_WM_CREATE()
025:     //}}AFX_MSG_MAP
026: END_MESSAGE_MAP()
027:
028: static UINT indicators[] =
```

```
029: {
030:     ID_SEPARATOR,           // status line indicator
031:     ID_INDICATOR_CAPS,
032:     ID_INDICATOR_NUM,
033:     ID_INDICATOR_SCRL,
034: };
035:
036: /////////////////////////////////////////////////
037: // CMainFrame construction/destruction
038:
039: CMainFrame::CMainFrame()
040: {
041:     // TODO: add member initialization code here
042:
043: }
044:
045: CMainFrame::~CMainFrame()
046: {
047: }
048:
049: int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
050: {
051:     if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
052:         return -1;
053:
054:     if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
          WS_VISIBLE | CBRS_TOP
055:         | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
          CBRS_SIZE_DYNAMIC) ||
056:         !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
057:     {
058:         TRACE0("Failed to create toolbar\n");
059:         return -1;      // fail to create
060:     }
061:
062:     if (!m_wndStatusBar.Create(this) ||
063:         !m_wndStatusBar.SetIndicators(indicators,
064:         sizeof(indicators)/sizeof(UINT)))
065:     {
066:         TRACE0("Failed to create status bar\n");
067:         return -1;      // fail to create
068:     }
069:
```

```
070:     // TODO: Delete these three lines if you don't want the toolbar
         // to
071:     //  be dockable
072:     m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
073:     EnableDocking(CBRS_ALIGN_ANY);
074:     DockControlBar(&m_wndToolBar);
075:
076:     return 0;
077: }
078:
079: BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
080: {
081:     if( !CFrameWnd::PreCreateWindow(cs) )
082:         return FALSE;
083:     // TODO: Modify the Window class or styles here by modifying
084:     //  the CREATESTRUCT cs
085:
086:     return TRUE;
087: }
088:
089: /////////////////////////////////////////////////
090: // CMainFrame diagnostics
091:
092: #ifdef _DEBUG
093: void CMainFrame::AssertValid() const
094: {
095:     CFrameWnd::AssertValid();
096: }
097:
098: void CMainFrame::Dump(CDumpContext& dc) const
099: {
100:     CFrameWnd::Dump(dc);
101: }
102:
103: #endif //_DEBUG
104:
105: /////////////////////////////////////////////////
106: // CMainFrame message handlers
```

DECLARE_DYNCREATE()에 대응하는 매크로인 IMPLEMEN_DYNCREATE()를 18번 줄에서 발견할 수 있다.

```
018: IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
```

MFC의 IMPLEMENT_DYNCREATE는 AFX.H의 830번 줄에서 찾아 볼 수 있다.

```
#define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
    CObject* PASCAL class_name::CreateObject() \
        { return new class_name; } \
    IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, \
        class_name::CreateObject)
```

예상대로 CreateObject()의 정의가 있다. 또한, 이것은 또한 811번 줄의 IMPLEMENT_RUNTIMECLASS 매크로를 호출한다.

```
#define IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema,\
 pfnNew) \
    AFX_COMDAT const AFX_DATADEF CRuntimeClass\
 class_name::class##class_name = { \
        #class_name, sizeof(class class_name), wSchema, pfnNew, \
            RUNTIME_CLASS(base_class_name), NULL }; \
    CRuntimeClass* class_name::GetRuntimeClass() const \
        { return RUNTIME_CLASS(class_name); } \
```

IMPLEMENT_RUNTIMECLASS는 Runtime Class와 관계된 클래스 정적 멤버 변수를 초기화하며, CRuntimeClass의 정보를 얻는 가상함수 GetRuntimeClass()를 정의한다. 이 매크로에 대응하는 DECLARE_RUNTIMECLASS는 존재하지 않는다.

```
024:     ON_WM_CREATE()
```

24번 줄에서 CMainFrame에 처음으로 유일하게 등장하는 메시지 매크로를 찾아 볼 수 있다. 이 매크로는 윈도우 메시지 WM_CREATE에 대응하는 매크로이다. 이러한 매크로의 종류는 몇 가지나 될까?

시그너처(signature)

이벤트(event)에 관계된 메시지의 매크로는 AFXMSG_.H의 136번 줄부터

찾아 볼 수 있다. 매크로는 모두 ON_...()으로 시작하는데, 이것은 이벤트 발생시 불리는 이벤트 핸들러임을 의미한다. 이 중, ON_COMMAND...(), ON_UPDATE_COMMAND...(), ON_NOTIFY...(), ON_CONTROL...(), ON_STN...(), ON_EN...(), ON_BN...(), ON_LBN...()와 ON_CBN...()으로 시작하는 매크로들은 실제로 윈도우의 메시지와 일대 일 대응되는 매크로가 아니다. 윈도우의 메시지와 일대 일 대응하는 매크로는 모두 ON_WM...()으로 시작한다. 이것은 윈도우 메시지가 모두 WM_...로 시작하기 때문이다.

예를 들어, WM_CREATE메시지를 살펴보자. 윈도우 메시지를 받는 **윈도우 프로시저(window procedure)** 함수가 운영체제부터 4개의 파라미터를 받는다는 것을 상기하자. 그것은 다음과 같다.

```
HWND hWnd,  // handle to window
UINT msg,   // message identifier
WPARAM wParam,// first message parameter
LPARAM lParam // second message parameter
```

hWnd는 메시지를 받는 윈도우에 대한 핸들이다. msg에는 WM_로 시작하는 2바이트 정수가 넘어온다. 예를 들면, 윈도우 생성시에는 WM_CREATE의 값이 msg에 넘어올 것이다. 마지막 두 개의 파라미터에 주목하자. 하나는 워드 파라미터라고 알려진 wParam이며, 다른 하나는 롱 파라미터(long parameter)로 알려진 lParam이다. 이 두 개의 파라미터는 메시지에 의존적인 값으로 설정된다. 즉, 메시지의 종류에 따라, 해석하는 방법이 달라진다.

WM_CREATE의 경우 wParam은 사용되지 않으며, lParam에는 CREATESTRUCT cs에 대한 포인터가 넘어온다. 그러므로, lParam을 다음과 같이 타입 변환하여 사용해야 한다.

```
lpcs = (LPCREATESTRUCT) lParam; // structure with creation data
```

WM_CREATE에 대응하는 ON_WM_CREATE() 매크로를 살펴보자. 그것은 다음과 같다.

```
#define ON_WM_CREATE() \
    { WM_CREATE, 0, 0, 0, AfxSig_is, \
        (AFX_PMSG)(AFX_PMSGW)(int (AFX_MSG_CALL\
 CWnd::*)(LPCREATESTRUCT))&OnCreate },
```

매크로를 살펴보면, 구조체의 처음 멤버는 WM_CREATE이며, 마지막 멤버는 OnCreate()의 시작 주소(&OnCreate)이다. 이것은 메시지가 WM_CREATE이며, OnCreate()함수가 호출될 것을 지시한다. 또한 OnCreate()의 파라미터는 lParam이 되어야 하며, 또한 이것은 LPCREATESTRUCT로 해석되어야 한다. 이것은 어떻게 해석하는 것일까? 파라미터 해석의 비밀은 다섯 번째 파라미터인 상수 **AfxSig_is**에 숨어 있다. 이것을 찾아보자.

AfxSig_is는 WINCORE.CPP의 1590번 줄부터 정의된 멤버 함수 OnWndMsg()의 switch문에서 발견할 수 있다. 이것은 1810번 줄에 위치한다.

```
    case AfxSig_is:
        lResult = (this->*mmf.pfn_is)((LPTSTR)lParam);
        break;
```

예상대로 이것은 lParam을 파라미터로 전달한다. 즉 OnCreate()의 파라미터로 전달되는 것이다. lResult는 pResult(OnWndMsg()의 마지막 파라미터)로 OnWndMsg()를 호출한 함수에 전달될 것이다.

이와 같이 MFC는 메시지 핸들러가 wParam과 lParam을 해석하는 방법에 따라 메시지들의 종류를 구분하여 서로 다른 프로토타입의 메시지 핸들러를 사용한다. 이것을 메시지의 **시그너처(signature)**라고 한다.

AFXMSG_.H 파일의 45번 줄에서 enum으로 정의된 시그너처 상수들을 확인할 수 있다. 우리는 API구현에서 시그너처를 사용하지 않았지만, MFC의 메시지 맵의 엔트리는 이 값들 중 하나를 반드시 사용한다. 시그너처가 파라미터를 해석하는 방법은 각 시그너처에 주석으로 명시되어 있다.

```
enum AfxSig
{
    AfxSig_end = 0,     // [marks end of message map]

    AfxSig_bD,      // BOOL (CDC*)
    AfxSig_bb,      // BOOL (BOOL)
    AfxSig_bWww,    // BOOL (CWnd*, UINT, UINT)
    AfxSig_hDWw,    // HBRUSH (CDC*, CWnd*, UINT)
    AfxSig_hDw,     // HBRUSH (CDC*, UINT)
    AfxSig_iwWw,    // int (UINT, CWnd*, UINT)
    AfxSig_iww,     // int (UINT, UINT)
    AfxSig_iWww,    // int (CWnd*, UINT, UINT)
    AfxSig_is,      // int (LPTSTR)
```

```
    AfxSig_lwl,     // LRESULT (WPARAM, LPARAM)
    AfxSig_lwwM,    // LRESULT (UINT, UINT, CMenu*)
    AfxSig_vv,      // void (void)

    AfxSig_vw,      // void (UINT)
    AfxSig_vww,     // void (UINT, UINT)
    AfxSig_vvii,    // void (int, int) // wParam is ignored
    AfxSig_vwww,    // void (UINT, UINT, UINT)
    AfxSig_vwii,    // void (UINT, int, int)
    AfxSig_vwl,     // void (UINT, LPARAM)
    AfxSig_vbWW,    // void (BOOL, CWnd*, CWnd*)
    AfxSig_vD,      // void (CDC*)
    AfxSig_vM,      // void (CMenu*)
    AfxSig_vMwb,    // void (CMenu*, UINT, BOOL)

    AfxSig_vW,      // void (CWnd*)
    AfxSig_vWww,    // void (CWnd*, UINT, UINT)
    AfxSig_vWp,     // void (CWnd*, CPoint)
    AfxSig_vWh,     // void (CWnd*, HANDLE)
    AfxSig_vwW,     // void (UINT, CWnd*)
    AfxSig_vwWb,    // void (UINT, CWnd*, BOOL)
    AfxSig_vwwW,    // void (UINT, UINT, CWnd*)
    AfxSig_vwwx,    // void (UINT, UINT)
    AfxSig_vs,      // void (LPTSTR)
    AfxSig_vOWNER,  // void (int, LPTSTR), force return TRUE
    AfxSig_iis,     // int (int, LPTSTR)
    AfxSig_wp,      // UINT (CPoint)
    AfxSig_wv,      // UINT (void)
    AfxSig_vPOS,    // void (WINDOWPOS*)
    AfxSig_vCALC,   // void (BOOL, NCCALCSIZE_PARAMS*)
    AfxSig_vNMHDRpl,    // void (NMHDR*, LRESULT*)
    AfxSig_bNMHDRpl,    // BOOL (NMHDR*, LRESULT*)
    AfxSig_vwNMHDRpl,   // void (UINT, NMHDR*, LRESULT*)
    AfxSig_bwNMHDRpl,   // BOOL (UINT, NMHDR*, LRESULT*)
    AfxSig_bHELPINFO,   // BOOL (HELPINFO*)
    AfxSig_vwSIZING,    // void (UINT, LPRECT) -- return TRUE

    // signatures specific to CCmdTarget
    AfxSig_cmdui,   // void (CCmdUI*)
    AfxSig_cmduiw,  // void (CCmdUI*, UINT)
    AfxSig_vpv,     // void (void*)
    AfxSig_bpv,     // BOOL (void*)
```

```
    // Other aliases (based on implementation)
    AfxSig_vwwh,                // void (UINT, UINT, HANDLE)
    AfxSig_vwp,                 // void (UINT, CPoint)
    AfxSig_bw = AfxSig_bb,      // BOOL (UINT)
    AfxSig_bh = AfxSig_bb,      // BOOL (HANDLE)
    AfxSig_iw = AfxSig_bb,      // int (UINT)
    AfxSig_ww = AfxSig_bb,      // UINT (UINT)
    AfxSig_bv = AfxSig_wv,      // BOOL (void)
    AfxSig_hv = AfxSig_wv,      // HANDLE (void)
    AfxSig_vb = AfxSig_vw,      // void (BOOL)
    AfxSig_vbh = AfxSig_vww,    // void (BOOL, HANDLE)
    AfxSig_vbw = AfxSig_vww,    // void (BOOL, UINT)
    AfxSig_vhh = AfxSig_vww,    // void (HANDLE, HANDLE)
    AfxSig_vh = AfxSig_vw,      // void (HANDLE)
    AfxSig_viSS = AfxSig_vwl,   // void (int, STYLESTRUCT*)
    AfxSig_bwl = AfxSig_lwl,
    AfxSig_vwMOVING = AfxSig_vwSIZING,  // void (UINT, LPRECT) --
                                        // return TRUE

    AfxSig_vW2,                 // void (CWnd*) (CWnd* comes
                                // from lParam)
    AfxSig_bWCDS,               // BOOL (CWnd*, COPYDATASTRUCT*)
    AfxSig_bwsp,                // BOOL (UINT, short, CPoint)
    AfxSig_vws,
};
```

이제 MainFrame의 49번 줄부터 정의된 OnCreate()를 보자.

```
049: int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

첫 번째 파라미터가 LPCREATESTRUCT 타입인 것을 알 수 있다.

이제 다른 중요한 메시지인 WM_PAINT에 대응하는 메시지 매크로인 ON_WM_PAINT를 살펴보자. 이것은 afxmsg_.h의 583번 줄에 있다.

```
#define ON_WM_PAINT() \
    { WM_PAINT, 0, 0, 0, AfxSig_vv, \
        (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL\
        CWnd::*)(void))&OnPaint },
```

이 매크로는 WM_PAINT에 대응하는 함수로 OnPaint()가 불려질 것을 지정한다. WM_PAINT에서 wParam와 lParam은 어떻게 해석될까?

```
hdc = (HDC) wParam; // the device context to draw in
```

lParam은 사용되지 않으며, wParam은 그리고자 하는 표면의 디바이스 컨텍스트(DC)에 대한 핸들이다. AfxSig_vv가 이것을 해결하고 있을 것이다.

WM_으로 시작하는 윈도우 메시지는 약 200여 개가 넘는다(WM_ACTIVE부터 WM_WININICHANGE까지). 또한, 이들 메시지에서 wParam과 lParam을 해석하는 방법은 다르다. 그러므로, 해당하는 MFC 이벤트 핸들러의 파라미터를 설정하는 방법도 메시지마다 달라야 할 것이다. 그래서 WM_으로 시작하는 각 메시지들을 시그너쳐로 구분하여 매크로를 만든 것이다. 윈도우 메시지에 대응하는 매크로는 일관성을 가진다. 그것들의 예는 아래와 같다.

```
WM_CREATE           ON_WM_CREATE()
WM_PAINT            ON_WM_PAINT()
WM_LBUTTONDOWN      ON_WM_LBUTTONDOWN
WM_KEYDOWN          ON_WM_KEYDOWN
WM_TIMER            ON_WM_TIMER
```

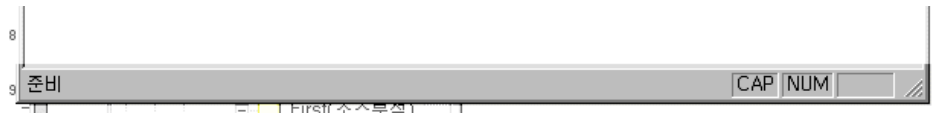
그러면, ON_BN_,..(), ON_COMMAND...() 등은 어떤 메시지에 대응하는 매크로일까? 이것은 자식 윈도우(child window)가 부모에게 보내는 통보 메시지, 혹은 메뉴 명령 메시지의 매크로이다. 실제로 WM_COMMAND 메시지는 메뉴에 의해서도 자식 윈도우에 의해서도 발생한다. 또한, WM_COMMAND 메시지의 wParam과 lParam을 해석하는 방법은 다르다. 그래서 MFC에서 이러한 종류의 메시지에 대해 별도의 매크로를 준비하여 둔 것이다. 우리는 ON_COMMAND...()등의 매크로를 메뉴(menu)와 대화상자(dialog box)를 다룰 때 다시 볼 것이다.

28~34번 줄은 상태바(status bar)의 지시자(indicator)들이다.

```
028: static UINT indicators[] =
029: {
030:     ID_SEPARATOR,           // status line indicator
031:     ID_INDICATOR_CAPS,
032:     ID_INDICATOR_NUM,
```

```
033:     ID_INDICATOR_SCRL,
034: };
```

위의 설정은 다음 [그림 11.3]과 같이 상태바를 윈도우의 하단에 표시한다.



[그림 11.3] 기본 상태바: 상태바도 하나의 윈도우(one window)이다.

상태바 배열의 초기값을 한 개 추가하여, 다음과 같이 바꾸어 보자.

```
028: static UINT indicators[] =
029: {
030:     ID_SEPARATOR,           // status line indicator
031:     ID_SEPARATOR,           // status line indicator
032:     ID_INDICATOR_CAPS,
033:     ID_INDICATOR_NUM,
034:     ID_INDICATOR_SCRL,
035: };
```

그러면 상태바는 다음과 같이 나타날 것이다.

[그림 11.4] 상태바 지시자의 추가: 상태바의 모양이 바뀌었다.

물론 상태바에 비트맵을 그린다든지 하는 등의 작업은 CStatusBar의 멤버 함수와 이벤트 핸들러를 사용해야 한다. 독자들은 여기서 단지 상태바의 초기 설정이 28번 줄에서 명시된다는 사실을 명심하라. 또한, 배열의 크기 명시 없이 모든 요소를 참조해야 하므로, 64번 줄에서처럼

```
064:          sizeof(indicators)/sizeof(UINT)))
```

sizeof-연산자를 사용하여 배열의 크기를 구하는 것이 올바른 코딩 방법이다.

이제 ON_WM_CREATE() 매크로에 의해 호출되는 함수 OnCreate()를 살펴보자. 이것은 49번 줄부터 정의되어 있다.

```
049: int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
050: {
```

OnCreate()는 LPCREATESTRUCT를 파라미터로 받는다. WM_CREATE는 화면에 표시할 윈도우를 유지할 **윈도우 구조체(window structure)**를 메모리에 할당한 후, 발생한다. 그러므로, 아직 화면에는 윈도우가 그려지지 않았다는 사실을 주의해야 한다. 즉, 최초의 그리기 작업을 WM_CREATE에서 하는 것은 불가능하다(그릴 곳이 없으므로).

```
051:     if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
052:         return -1;
```

OnCreate()는 먼저 상위 클래스의 OnCreate()를 호출한다.

```
054:     if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |\
 WS_VISIBLE | CBRS_TOP
055:         | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |\
 CBRS_SIZE_DYNAMIC) ||
056:         !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
```

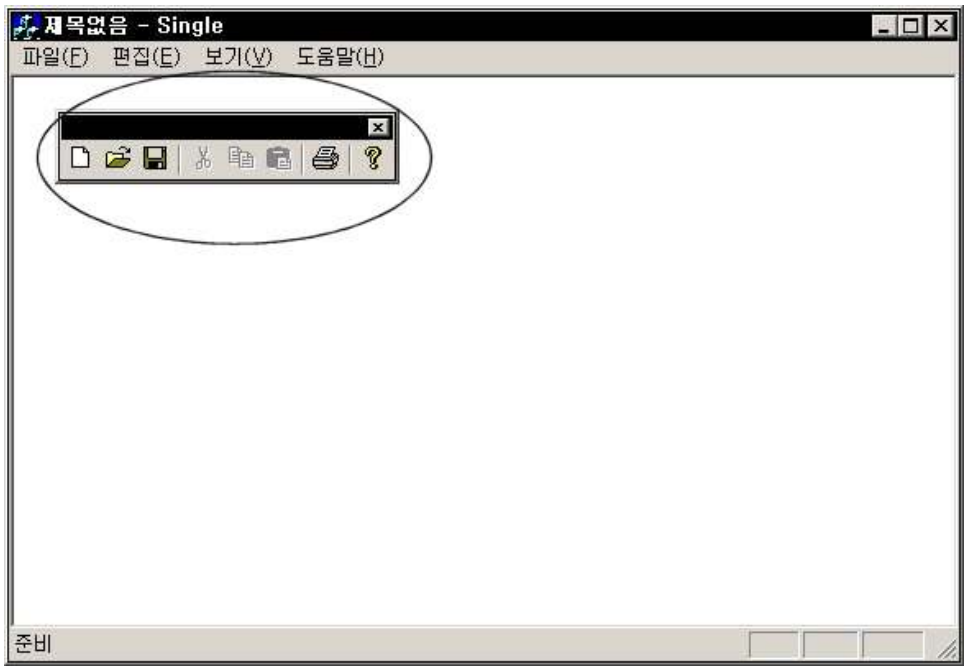
그리고 54번 줄에서 도구바를 만들고, 해당 비트맵을 로드한다.

```
062:     if (!m_wndStatusBar.Create(this) ||
063:         !m_wndStatusBar.SetIndicators(indicators,
064:           sizeof(indicators)/sizeof(UINT)))
```

다음은 62번 줄에서 상태바를 만들고, 설정한다.

```
072:     m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
073:     EnableDocking(CBRS_ALIGN_ANY);
074:     DockControlBar(&m_wndToolBar);
```

기본 도구바는 MainFrame과 도킹(docking)할 수 있다. 이 설정이 72~74번 줄에서 이루어진다. 각각을 도킹 가능하도록 만든 다음, 74번 줄에서 실제 도킹을 한다.



[그림 11.5] 도구바의 도킹: 도구바는 메인 프레임의 가장자리에 도킹 가능하도록 설정된다.

이제 PreCreateWindow()를 79~87번 줄에서 발견할 수 있다.

```
079: BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
080: {
081:     if( !CFrameWnd::PreCreateWindow(cs) )
082:         return FALSE;
083:     // TODO: Modify the Window class or styles here by modifying
084:     //  the CREATESTRUCT cs
085:
086:     return TRUE;
087: }
```

PreCreateWindow()는 이름이 의미하듯이, OnCreate()가 불리기 전에 반드시 불린다. 하지만, 이 함수는 윈도우 메시지에 대응하는 이벤트 핸들러가 아니므로, 클래스 위저드에 의해서 가상함수로 관리된다는 것을 다시 한번 상기하자. MainFrame.h의 28번 줄의 선언을 다시 보자.

```
027:     //{{AFX_VIRTUAL(CMainFrame)
028:     virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
029:     //}}AFX_VIRTUAL
```

PreCreateWindow()가 가상 함수로 관리된다는 것을 확인할 수 있다. 이 함수는 OnCreate()가 받는 구조체와 같은 구조체의 참조(reference)를 파라미터로 받는다. 그러므로 윈도우가 그려지기 전에 윈도우의 모양을 설정하기 위해서는 이 함수를 적절하게 변경해야 한다. 84번 줄에 다음 줄을 추가해 보자.

```
081:     if( !CFrameWnd::PreCreateWindow(cs) )
082:         return FALSE;
083:     // TODO: Modify the Window class or styles here by modifying
         cs.cx=200;
         cs.cy=200;
086:     //  the CREATESTRUCT cs
```

이제 프로그램을 실행해 보면, 윈도우의 가로 세로 크기가 200×200으로 설정된 것을 확인할 수 있다.

92번 줄과 103번 줄 사이에 정의된 2개의 함수는 디버깅을 위해 존재한다.

```
092: #ifdef _DEBUG
093: void CMainFrame::AssertValid() const
094: {
095:     CFrameWnd::AssertValid();
096: }
097:
098: void CMainFrame::Dump(CDumpContext& dc) const
099: {
100:     CFrameWnd::Dump(dc);
101: }
102:
103: #endif //_DEBUG
```

만약 빌드 타겟이 릴리즈(release)라면 AssertValid()와 Dump()는 단순하게 무시될 것이다. 이 함수들은 자신이 유효한 상태(메모리 할당이 되었는지 등)인지를 검사하기 위한 디버깅 함수이다. 물론 사용자가 ASSERT_VALID 매크로를 사용하여, 이 함수를 호출할 수도 있다. ASSERT_VALID는 AFX.H의

228번 줄에서 찾을 수 있다.

```
#define ASSERT_VALID(pOb)  (::AfxAssertValidObject(pOb, THIS_FILE,\
 __LINE__))
```

OBJCORE.CPP의 72번 줄에 AfxAssertValidObject()의 정의를 찾을 수 있다.

[예제 11.3] AfxAssertValidObject()

```
#ifdef _DEBUG
void AFXAPI AfxAssertValidObject(const CObject* pOb,
    LPCSTR lpszFileName, int nLine)
{
    if (pOb == NULL)
    {
        TRACE0("ASSERT_VALID fails with NULL pointer.\n");
        if (AfxAssertFailedLine(lpszFileName, nLine))
            AfxDebugBreak();
        return;     // quick escape
    }
    if (!AfxIsValidAddress(pOb, sizeof(CObject)))
    {
        TRACE0("ASSERT_VALID fails with illegal pointer.\n");
        if (AfxAssertFailedLine(lpszFileName, nLine))
            AfxDebugBreak();
        return;     // quick escape
    }

    // check to make sure the VTable pointer is valid
    ASSERT(sizeof(CObject) == sizeof(void*));
    if (!AfxIsValidAddress(*(void**)pOb, sizeof(void*), FALSE))
    {
        TRACE0("ASSERT_VALID fails with illegal vtable pointer.\n");
        if (AfxAssertFailedLine(lpszFileName, nLine))
            AfxDebugBreak();
        return;     // quick escape
    }

    if (!AfxIsValidAddress(pOb, pOb->GetRuntimeClass()->m_nObjectSize,\
 FALSE))
```

```
    {
        TRACE0("ASSERT_VALID fails with illegal pointer.\n");
        if (AfxAssertFailedLine(lpszFileName, nLine))
            AfxDebugBreak();
        return;     // quick escape
    }
    pOb->AssertValid();
}
```

AfxAssertValidObject()는 객체의 상태에 따라 몇 개의 디버깅 메시지를 출력하고, 마지막에 AssertValid()를 호출한다.

진단 매크로(diagnostic macros)

자신이 할당한 메모리가 유효한지를 검사하는 것은 좋은 습관이다. 이러한 타당성 검사를 위해서 몇 개의 매크로가 존재한다.

```
ASSERT(booleanExpression)
ASSERT_KINDOF(classname, pObject)
ASSERT_VALID(pObject)
DEBUG_NEW
TRACE(expression)
VERIFY(booleanExpression)
```

① ASSERT(booleanExpression)

booleanExpression을 검사하여, 메시지를 출력한다. 만약, booleanExpression이 0이면, 프로그램은 종료한다. 아래의 예를 참조하라.

```
// example for ASSERT
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject.
ASSERT( pcage!= NULL )
ASSERT( pcage->IsKindOf( RUNTIME_CLASS( CAge ) ) )
// Terminates program only if pcage is NOT a CAge*.
```

② ASSERT_KINDOF(classname, pObject)

classname은 CObject를 상위 클래스로 가지는 클래스 이름이다. pObject에

는 객체의 주소를 명시한다. pObject가 classname 타입이면, TRUE가 된다.

```
ASSERT_KINDOF(CMyDocument, pDocument)
```

는

```
ASSERT(pobject->IsKindOf(RUNTIME_CLASS(classname)
```

와 동일하다. 즉 클래스의 CRuntimeClass의 클래스 이름에 대해 스트링 검사를 수행하는 것이다.

③ ASSERT_VALID(pObject)

AfxAssertValidObject를 호출한다. 이 함수는 내부에서 AssertValid()를 호출한다. 독자들은 93번 줄의

```
093: void CMainFrame::AssertValid() const
```

를 기억할 것이다. 즉, 필요한 동작을 AssertValid()에 기술하고 ASSERT_VALID()를 호출하면 된다.

④ DEBUG_NEW

빌드 타겟이 디버그인 경우 연산자 new는 DEBUG_NEW로 대치된다. 이 매크로는 AFX.H의 1632번 줄에서 찾을 수 있다.

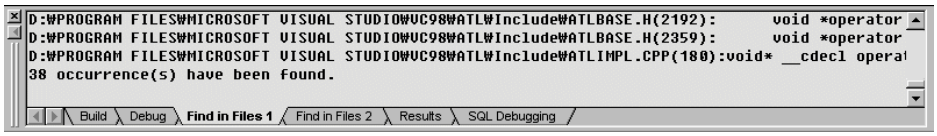
```
void* AFX_CDECL operator new(size_t nSize, LPCSTR lpszFileName, int\
 nLine);
#define DEBUG_NEW new(THIS_FILE, __LINE__)
```

DEBUG_NEW은 오버로딩된 연산자 new 함수를 호출한다. 이 함수에서는 메모리 할당이외의 추가적인 디버깅 작업을 수행한다.

⑤ TRACE(expression)

expression을 Output 윈도우에 출력한다. 다음의 예를 참고하라.

```
// example for TRACE
int i = 1;
char sz[] = "one";
TRACE( "Integer = %d, String = %s\n", i, sz );
// Output: 'Integer = 1, String = one'
```



[그림 11.6] TRACE() 매크로: TRACE()매크로는 printf()처럼 디버깅 정보를 Output 윈도우에 출력할 수 있다.

⑥ VERIFY(booleanExpression)

ASSERT()와 같지만, 릴리즈 모드에서도 동작한다. ASSERT()는 디버그 모드에서만 동작하도록 정의되어 있다.

요약

이제 Single 프로젝트의 CMainFrame의 소스 분석을 마쳤다. 이 장까지의 내용을 모두 이해한 독자들은 남은 나머지 두 장을 이해하는 것은 시간문제이다. DDX의 자세한 사항을 12장에서 다룰 것이다. 또한, 13장에서 직렬화의 분석을 할 것이며, 그 때 CSingleDoc과 CSingleView의 소스를 자세히 분석할 것이다.

- Single 프로젝트의 **MainFrame** 소스를 살펴 보았다.
- 함수의 시그너쳐란 함수의 프로토타입을 말하는 것인데, **메시지 맵의 시그너쳐**는 메시지 맵에서 메시지 핸들러의 파라미터 타입을 결정하는 ID이다.

[문서의 끝]