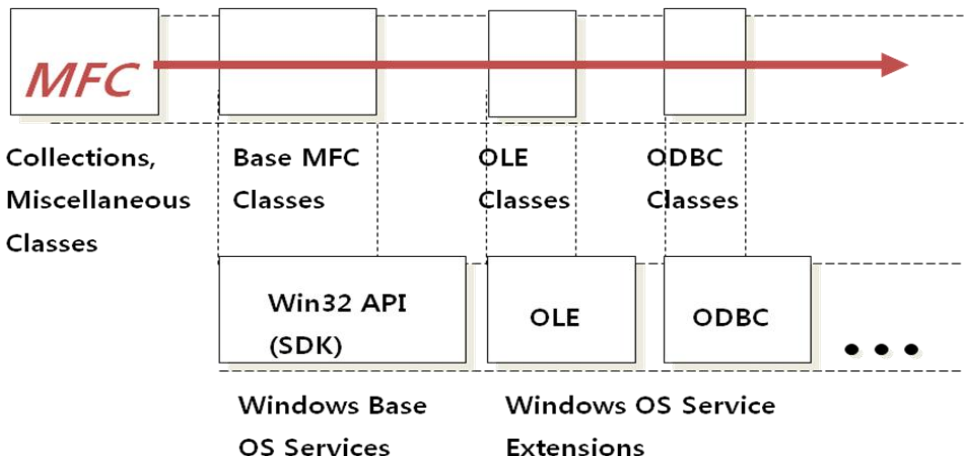
1. MFC(Microsoft Foundation Class)란 무엇인가?

1.1 MFC란 무엇인가?

윈도우 프로그램을 작성하기 위해 윈도우 API라는 것을 사용한다. 라이브러리나 API라는 것은 여러 가지 유용한 함수들의 집합체를 말한다. MFC는 단순한 라이브러리가 아닌 유용한 클래스들의 집합체이다. 클래스라는 것은 함수(맴버 함수)만 있는 것이 아니라 해당 함수들과 연관된 데이터(맴버 변수)를 함께 포함하고 있기 때문에 일반 라이브러리와는 구별된다.

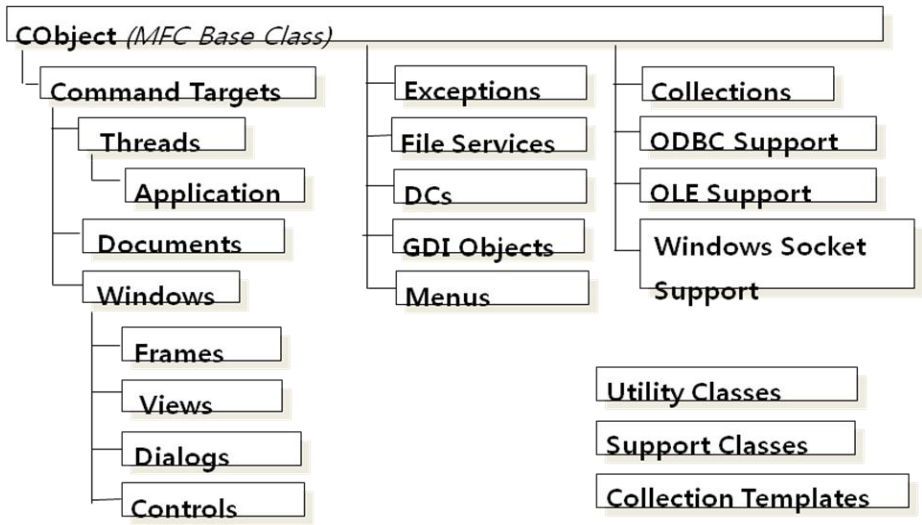
그래서 <클래스 라이브러리>라는 용어를 사용하는데 VisualC++이 제공하는 클래스 라이브러리가 바로 MFC이다.

MFC 프로그래밍을 배우는 과정은 C++에 대한 기본 지식과 윈도우즈 프로그래밍에 대한 이해를 바탕으로 MFC가 제공하는 클래스들의 종류와 기능, 구조를 익혀나가는 과정이다. 또한 MFC자체가 윈도우즈 API를 근간으로 하기 때문에 SDK 프로그래밍에 대한 경험이 있으면 MFC를 익히는데 커다란 도움이 된다.



MFC 라이브러리는 윈도우즈 API 이외에도 OLE, ODBC, 윈속 관련 클래스를 포함하여 230여개의 클래스를 제공한다. MSDN을 살펴보면 MFC의 전체 계층 구조도를 볼 수 있다. 아래는 계층 구조의 대략적인 구조를 표한한 그림이다.

이제 MFC 프로그래밍을 이해해 보도록 하자.



1.2 에플리케이션 프레임워크(Applicaiton Framework) 특징

어플리케이션 프레임워크(이하 프레임워크)는 AppWizard, 클래스위저드(ClassWizard), 워크스페이스 등과 MFC를 유기적으로 연결하여 응용 프로그램을 보다 쉽고 편리하게 만들어주는 도구이다. 일반 라이브러리는 특정 프로그램 안에 편입되어 사용되도록 고안되었지만 프레임워크는 프로그램 구조(골격) 자체를 정의한다.

예를 들어 메시지 박스를 하나 띄우기 위해 SDK기반으로 프로그래밍을 하면 수십 라인을 작성해야 한다. 구조를 살펴보면 실제 메시지 박스를 띄우는 부분은 길어야 1,2 라인 정도지만 윈도우를 등록/생성/기본 처리하는 부분이 나머지를 차지한다. 이것은 다른 SDK 프로그램 코드에서도 흔 히 볼 수 있는 부분으로 누가 작성하더라도 거의 유사한 정형화

된 코드들이다.

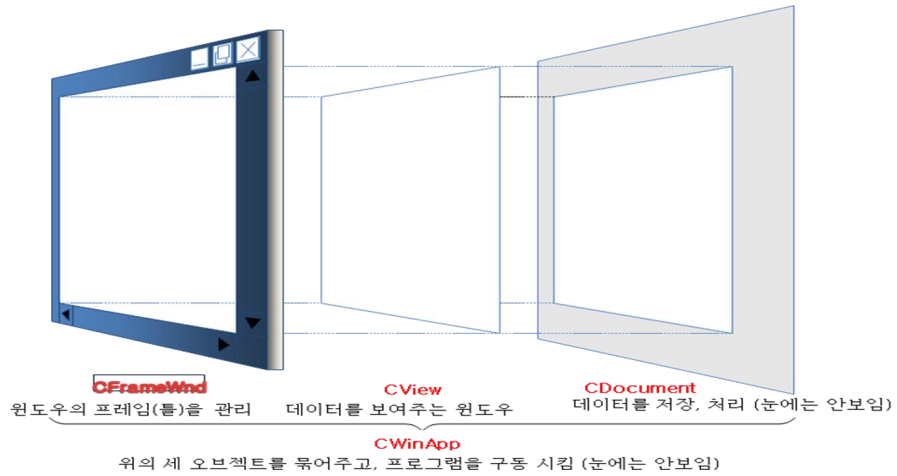
이제는 App Wizard를 이용하여 프로그램의 기본 골격 코드를 자동으로 생성할 수 있고 클래스 위저드나 속성 페이지, 워크스페이스 등을 이용하여 코드를 보다 쉽게 작성할 수 있게 되었다.

어플리케이션 프레임워크를 사용하여 얻는 이점은 다음과 같다.

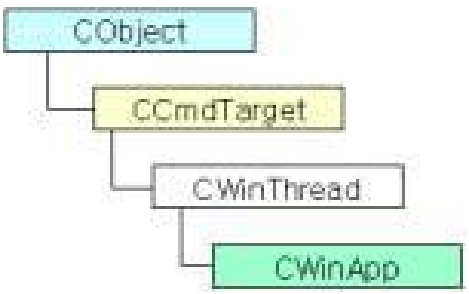
- 프레임워크의 어플리케이션은 표준화된 구조를 사용한다.
  유지/보수가 용이하다.
  조각난 프로젝트를 엮어나가기가 쉽다.
- 프레임워크의 애플리케이션은 작고 빠르다.
  프레임워크가 제공하는 기능은 DLL로 작성되었으므로 실행파일 크기는 작아 졌으며 DLL이 최적화된 컴파일러로 만들어진 기계 코드를 사용하므로 실행 속도가 빠르다.
- 프레임워크를 이용하면 코딩 작업이 줄어 개발 기간이 단축된다.
  AppWizard, ClassWizard, 리소스 편집기, 컴포넌트 갤러리 등은 일반적인 사용자 인터페이스 및 정형화된 코드를 작성하는데 필요한 코드 작업과 시간을 줄여준다.

1.3 에플리케이션 프레임워크(Applicaiton Framework) 이해

MFC에서 제공하는 Application Framework는 아래 그림에서 보이는 4개의 주요 클래스를 제공하고 있으며, 이와 관련된 다양한 클래스로 구성되어 있다. 이와 관련된 내용을 학습하고자 한다.

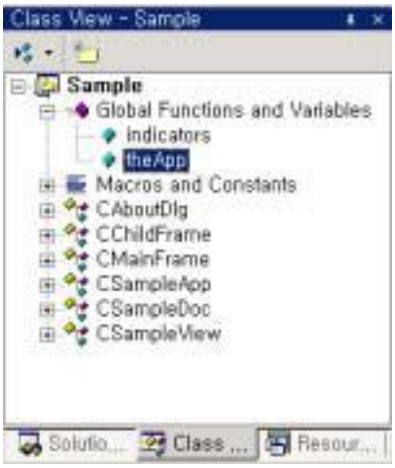


1.3.1.CWinApp : 응용 프로그램 클래스



CWinApp 클래스는 MFC 응용 프로그램의 초기화, 메시지 루프의 실행, 종료 등을 관리하는 응용 프로그램 클래스이다. MFC를 이용하여 작성한 응용 프로그램은 CWinApp 클래스에서 유도된 클래스를 반드시 그리고 오직 하나 가진다. 응용 프로그램 개체는 다른 전역 C++ 개체가 생성될 때 함께 생성되며 WinMain 함수가 호출되기 이전에 이미 생성되어 있다.

응용 프로그램 마법사가 생성한 MDI 형식의 응용 프로그램 골격을 살펴보면 전역 개체로 응용프로그램 개체가 선언되어 있음을 볼 수 있다.



CWinApp 클래스는 CWinThread 클래스의 유도 클래스로 응용 프로그램이 가질 수 있는 여러 쓰레드 중 메인 쓰레드를 구성한다. 응용 프로그램 개체의 주요 함수들인 InitInstance, Run, ExitInstance, OnIdle 등은 모두 CWinThread 클래스의 멤버 함수들이다.

다른 윈도우 기반 응용 프로그램과 마찬가지로 MFC로 작성한 응용 프로그램도 WinMain 함수를 가진다. 하지만 이 함수를 직접 작성할 필요는 없다. WinMain 함수는 MFC에서 제공해주며 응용 프로그램이 시작될 때 호출된다.

MFC의 WinMain함수는 윈도우 클래스를 등록하고 응용 프로그램을 초기화한 후 응용 프로그램을 시작합니다. WinMain 함수는 오버라이드할 수 없지만 WinMain 함수가 호출하는 CWinApp 클래스의 멤버 함수를 오버라이드함으로써 WinMain 함수의 동작을 수정할 수 있다.

응용 프로그램을 초기화하기 위해 WinMain 함수는 응용 프로그램 개체의 InitInstance 멤버 함수를 호출하고, 메시지 루프를 시작하기 위해서는 Run 멤버 함수를 호출한다. 응용 프로그램이 종료하는 경우에 WinMain 함수는 ExitInstance 멤버 함수를 호출한다. 다음 그림은 MFC 응용 프로그램 개체에서 호출되는 멤버 함수들을 순서대로 나타낸 것이다.

CWinApp 클래스의 멤버 함수와 별도로 MFC는 CWinApp 응용 프로그램 개체를 접근할 수 있는 전역 함수들을 제공한다.

- AfxGetApp : CWinApp 개체에 대한 포인터를 반환한다.
- AfxGetInstanceHandle : 현재 응용 프로그램 개체에 대한 핸들을 반환한다.
- AfxGetResourceHandle : 응용 프로그램의 리소스에 대한 핸들을 반환한다.
- AfxGetAppName : 응용 프로그램의 이름을 포함하고 있는 문자열에 대한 포인터를 반환 한다.

만약 CWinApp 개체에 대한 포인터를 가지고 있다면 *m_pszExeName* 멤버 변수를 통해 응용 프로그램의 이름을 얻어올 수 있다.

가) 주요 멤버 함수

CWinApp 클래스에는 몇 가지 오버라이드할 수 있는 중요한 멤버 함수들이 있다. 실제로 이들 멤버 함수는 CWinApp 클래스가 CWinThread 클래스의 멤버 함수를 오버라이드한 것이다.

이들 중 CWinApp 클래스가 반드시 오버라이드해야 하는 함수는 InitInstance 함수이고, 마법사는 응용 프로그램을 위한 CWinApp 유도 클래스에서 InitInstance 함수를 오버라이드하고 있다.

InitInstance 멤버 함수

응용 프로그램은 윈도우즈 시스템에서 한 번 이상 실행될 수 있다. WinMain 함수는 새로운 응용 프로그램의 인스턴스가 실행될 때마다 InitInstance 함수를 호출한다.

virtual BOOL InitInstance( );

반환값 : 초기화에 성공하면 0이 아닌 값을, 실패하면 0을 반환한다.

마법사가 생성한 InitInstance 함수는 다음 내용들을 포함하고 있다.

- 다큐먼트, 뷰, 프레임 윈도우를 연결하는 다큐먼트 템플릿을 생성한다.
- .INI 파일이나 레지스트리에서 응용 프로그램에 관한 정보를 로드한다.
- 하나 이상의 다큐먼트 템플릿을 등록한다.
- MDI 형식의 응용 프로그램의 경우 메인 프레임 윈도우를 생성한다.
- 명령행 인자를 처리하여 명령행에 지정된 파일을 열거나 빈 다큐먼트를 생성한다.

다음은 MDI 형식의 응용 프로그램에서 마법사가 생성한 InitInstance 함수이다.

Run 멤버 함수

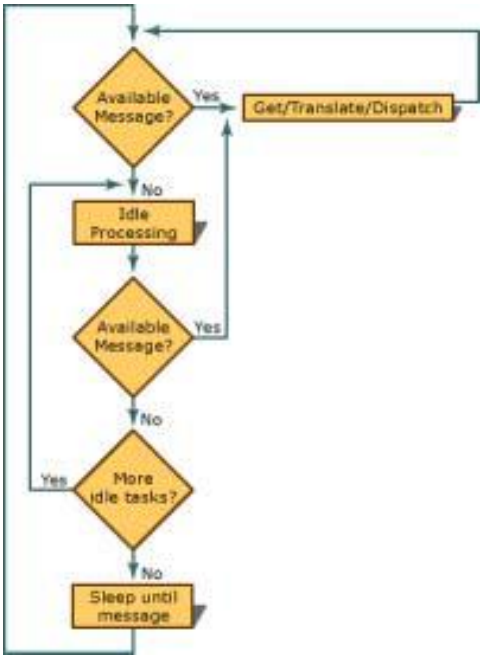
MFC 응용 프로그램은 CWinApp 클래스의 Run 멤버 함수에서 대부분의 시간을 소비한다.. 초기화 작업 이후 WinMain 함수는 메시지 루프를 위한 Run 멤버 함수를 호출한다.

virtual int Run( );

반환값 : WinMain 함수가 반환한 정수값을 반환한다.

Run 멤버 함수는 WM_QUIT 메시지를 읽어올 때까지 메시지 큐를 검사하고 메시지가 있는 경우에는 메시지 맵을 통해 메시지가 전달되도록 하며, 메시지가 없는 경우에는 OnIdle 함수를 호출하여 유휴 작업(idle job)을 처리한다. 처리할 메시지도 없고 유휴 작업도 없는 경우에는 응용 프로그램은 메시지가 발생될 때까지 응용 프로그램은 수면 상태에 있게 된다.

읽어온 메시지는 PreTranslateMessage 멤버 함수를 통해 필요한 처리를 한 후 표준 키보드 메시지 처리를 위해 TranslateMessage 함수로 전달된다. 마지막으로 DispatchMessage 함수가 호출된다. Run 함수를 오버라이드하여 특별한 처리를 추가할 수 있지만, 오버라이드되는 경우는 거의 없다.



ExitInstance 멤버 함수

ExitInstance 함수는 응용 프로그램의 인스턴스가 종료할 때 호출되며, 일반적으로 사용자가 응용 프로그램을 종료하고자 하는 경우에 호출된다.

virtual int ExitInstance( );

반환값 : 응용 프로그램의 종료 코드로 0은 오류가 없는 것을 나타낸다.
종료 코드는 WinMain 함수의 반환값으로 사용된다.

ExitInstance 함수를 오버라이드하여 그래픽 장치 인터페이스(GDI) 리소스를 해제하거나 프로그램 실행 중에 할당한 메모리를 해제할 수 있다. 다큐먼트나 뷰와 같은 표준 클래스의 해제는 기본 구현에서 제공하고 있으므로 오버라이드한 함수에서는 반드시 CWinApp 클래스의 ExitInstance 함수를 호출해주어야 한다.

OnIdle 멤버 함수

유휴 시간 작업을 수행하기 위해서 OnIdle 함수를 오버라이드할 수 있다.

virtual BOOL OnIdle(LONG lCount);

반환값 : 추가적인 유휴 작업이 필요하면 0이 아닌 값을 반환하고 더 이상의 유휴 작업이 필요하지 않으면 0을 반환한다.

Parameter	Description
lCount	응용 프로그램의 메시지 큐가 비어있을 때마다 OnIdle 함수가 호출 되고 이 때마다 증가되는 카운터를 나타낸다. 이 카운터는 새로운 메시지가 처리될 때 0으로 초기화된다. *lCount* 파라미터를 이용하면 메시지 처리 없이 유휴 상태에 있는 상대적인 시간의 길이를 알 수 있다.

작업이 필요한 경우에 이 카운터를 이용하면 여러 유휴 작업을 차례로 수행할 수 있으며, 유휴 작업의 수행 과정은 다음과 같다.

1) 메시지 큐에 메시지가 없으면 MFC는 *lCount* 파라미터를 0으로 설정하여 OnIdle 함수를 호출한다.

2) OnIdle 함수는 유휴 작업의 일부를 처리하고 추가적인 유휴 작업이 필요함을 알리기 위해 0이 아닌 값을 반환한다.

3) 메시지 루프가 메시지 큐를 다시 검사한다. 여전히 메시지가 없다면 *lCount* 파라미터를 증가시켜서 다시 OnIdle 함수를 호출한다.

4) OnIdle 함수가 유휴 작업을 끝내면 0을 반환한다. 이는 메시지 루프가 메시지 큐에서 다음 메시지를 받을 때까지 OnIdle 함수를 호출하지 않도록 한다. OnIdle 함수가 다시 호출될 때에 *lCount* 파라미터는 0으로 초기화된다. 사용자의 입력은 OnIdle 함수에서 반환하기 전까지 처리되지 않으므로 OnIdle 함수에서 오랜 시간이 걸리는 작업을 해서는 안 된다.

OnIdle 함수는 기본적으로 메뉴 항목이나 툴바 버튼과 같은 사용자 인터페이스 개체를

갱신하고 작업 중에 생성된 임시 개체들을 해제하도록 구현되어 있다. 따라서 OnIdle 함수를 오버라이드한 경우에는 오버라이드한 함수에서 *lCount*를 이용하여 CWinApp::OnIdle 함수를 호출해주어야 한다. 이 때 기반 클래스의 모든 유휴 작업이 끝날 때까지 즉, 기반 클래스의 OnIdle 함수가 0을 반환할 때까지 계속 호출해야 한다. 기반 클래스의 유휴 작업 처리가 끝나기 전에 수행할 작업이 있다면 카운터에 따라 유휴 시간을 나누어 사용하는 것이 바람직하다.

특정 메시지가 처리된 이후에는 OnIdle 메시지가 호출되지 않도록 하기 위해서는 CWinThread::IsIdleMessage 함수를 오버라이드하면 된다.

나) CWinApp 클래스의 부가 기능
응용 프로그램의 초기화, 메시지 루프의 실행, 종료 등을 관리하는 것 이외에도 응용 프로그램 개체는 몇 가지 부가적인 서비스를 제공한다. 자세한 사항은 MSDN을 참고하자

*** Shell Registration**
기본적으로 MFC 응용 프로그램 마법사는 윈도우 익스플로러나 파일 관리자에서 응용 프로그램이 작성한 데이터 파일을 더블 클릭으로 열 수 있는 기능을 제공한다.

*** 파일 관리자에서 끌어다 놓기**
Windows 3.1 이후 버전부터 윈도우즈 시스템은 끌어다 놓기(drag-and-drop)를 지원한다. 끌어다 놓기를 지원하기 위해서는 InitInstance 함수에서 메인 프레임 윈도우의 CWnd::DragAcceptFiles 멤버 함수를 호출해주면 된다.

```
pMainFrame->DragAcceptFiles();
```

위의 코드를 추가하면 MDI 형식의 응용 프로그램의 경우 익스플로러나 파일 관리자에서 끌어다 놓은 파일에 대해 새로운 다큐먼트 개체를 생성하여 차일드 윈도우를 통해 보여준다.

*** 최근 사용한 파일 목록**
사용자가 파일을 열고 닫을 때 응용 프로그램 개체는 최근 작업 파일들의 목록을 유지한

다. 이들 파일 이름은 파일 메뉴에 추가되어 관리된다. MFC는 이들 파일 이름을 레지스트리나 .INI 파일에 저장한다. InitInstance 함수에서 호출되고 있는 LoadStdProfileSettings 함수는 레지스트리나 .INI 파일에서 응용 프로그램에 관한 정보를 읽어오며, 여기에 최근 작업 파일에 관한 정보도 포함되어 있다.

LoadStdProfileSettings(4); // MRU를 포함하여 표준 INI 파일 옵션을 로드한다.

저장되는 최근 작업 파일의 수는 기본적으로 4개로 설정되어 있으며, 프로젝트를 생성할 때 그 값을 조정할 수 있다.

응용 프로그램의 정보는 윈도우의 버전에 따라 저장되는 방법이 다르다.

- Windows NT, 2000 이후 : 레지스트리에 저장된다.
- Windows 3.x : win.ini 파일에 저장된다.
- Windows 95 이후 : 캐시를 사용하는 win.ini 파일에 저장된다.

1.3.2 윈도우 개체(Window Object) : CWnd클래스와 관련되어 있다.

MFC는 윈도우에 대한 핸들인 HWND를 캡슐화한 클래스로 CWnd 클래스를 제공하고 있다.

CWnd 개체는 C++에서의 윈도우 개체와 동일한 것으로 윈도우를 나타내는 HWND를 포함하고 있다. CWnd 클래스는 윈도우가 가져야 하는 기본적인 기능들을 모두 포함하고 있는 클래스로 MFC의 모든 윈도우 개체들은 CWnd 클래스를 상속하여 만들어진다. CWnd 유도 클래스들로는 프레임 윈도우, 다이얼로그 박스, 차일드 윈도우, 컨트롤, 컨트롤바, 툴바 등이 있다.

MFC는 윈도우의 동작과 윈도우 관리를 위해 몇 가지 기본 구현을 제공하고 있다. 제공되는 기본 구현은 CWnd 클래스를 상속하여 수정할 수도 있지만, 대부분의 응용 프로그램 클래스들은 CWnd 클래스를 직접 상속하지 않고 그 유도 클래스를 상속하여 사용한다. CWnd 클래스는 모든 윈도우에 공통으로 적용되는 특성을 가진 클래스임을 생각해보면

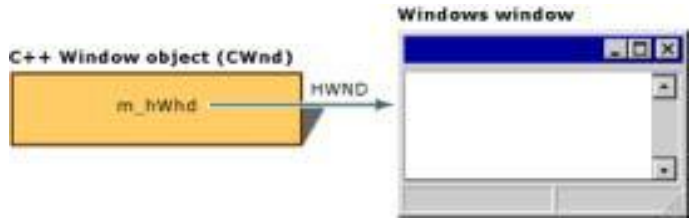
쉽게 이해할 수 있다.

CWnd 클래스와 그 유도 클래스들은 생성자와 소멸자와 더불어 개체를 초기화하고 윈도우의 내부 구조를 생성해주며 캡슐화된 HWND를 억세스할 수 있는 함수들을 제공하고 있다. 또한 메시지 보내기, 윈도우 상태 억세스, 좌표 변환, 업데이트, 스크롤, 클립보드 사용 등 윈도우 공통의 많은 작업을 위한 함수들을 제공한다. HWND를 파라미터로 가지는 대부분의 윈도우 API 함수들이 CWnd 클래스의 멤버 함수로 캡슐화되어 있다고 보면 된다. 이들 멤버 함수들은 HWND 파라미터를 제외하고는 API 함수와 대부분 동일한 파라미터를 가진다.

CWnd 클래스의 주요한 목적들 중 하나는 메시지 처리를 위한 인터페이스를 제공하는 것이다. CWnd 클래스는 CCmdTarget 클래스의 유도 클래스로 메시지 맵을 가지고 있다. 따라서 메시지 맵을 통해 메시지와 명령을 처리할 수 있다. CWnd 클래스의 많은 멤버 함수들은 표준 메시지들에 대한 핸들러들로 afx_msg를 함수 선언에 포함하고 있고 함수 이름은 관례에 따라 "On"으로 시작한다.

가) C++ 윈도우 개체와 HWND의 관계

"윈도우 개체"는 응용 프로그램이 생성한 CWnd 클래스 또는 그 유도 클래스의 개체를 말한다. 윈도우 개체는 생성자에 의해 생성되고 소멸자에 의해 파괴된다. 이에 비해 "윈도우"는 윈도우 자체를 나타내는 시스템 내부 데이터 구조에 대한 핸들로서 시스템 리소스를 소비한다. 윈도우는 윈도우 핸들(HWND)에 의해 구분되며 CWnd 클래스의 Create 함수에 의해 CWnd 개체가 생성된 이후 생성된다. 윈도우 개체가 나타내는 윈도우 즉, 윈도우 핸들은 윈도우 개체의 *m_hWnd* 멤버 변수에 저장되어 있다. 다음 그림은 윈도우 개체와 윈도우의 관계를 나타낸 것이다.



MFC에서 윈도우는 두 단계를 통해 생성되는 된다.

1) 먼저 윈도우 개체를 생성한다.
   이는 생성자를 통해 다른 C++ 개체와 마찬가지로 생성된다.
2) 윈도우 개체가 생성된 후 윈도우가 생성된다.
   윈도우의 생성은 윈도우 개체의 멤버 함수인 Create 함수를 통해 수행된다.

나) CWnd 유도 클래스

윈도우는 일반적으로 CWnd 클래스나 그 유도 클래스의 개체를 생성함으로써 생성할 수 있다. 하지만 MFC에서 사용되는 대부분의 윈도우는 직접 CWnd 클래스를 상속하지 않고 그 유도 클래스인 프레임 윈도우를 상속하여 만들어진다.

*** 프레임 윈도우 클래스**

CFrameWnd	단일 다큐먼트와 뷰를 포함하는 SDI 형식의 프레임 윈도우를 위해 사용된다. 이 프레임 윈도우는 응용 프로그램을 위한 메인 프레임 윈도우인 동시에 다큐먼트를 위한 프레임 윈도우의 역할을 한다.
CMDIFrameWnd	MDI 형식의 메인 프레임 윈도우를 위해 사용된다. 메인 프레임 윈도우는 모든 MDI 다큐먼트 윈도우를 포함하며 다큐먼트 윈도우들은 메인 프레임 윈도우의 메뉴를 공유한다. MDI 프레임 윈도우는 데스크탑에 나타나는 최상위 윈도우이다.
CMDIChildWnd	MDI 메인 프레임 윈도우에서 각각의 다큐먼트를 위해 사용된다. 각 다큐먼트와 뷰는 차일드 프레임에 연결되어 있다. MDI 차일드 윈도우는 메인 프레임 윈도우와 비슷하게 보이지만 메인 프레임 윈도우 내부에 포함되어 있으며, 자신의 메뉴바를 가지지 않고 자신을 포함하고 있는 메인 프레임 윈도우의 메뉴를 공유한다.

*** 기타 CWnd 유도 클래스**

프레임 윈도우 이외에도 다른 중요한 클래스들이 CWnd 클래스를 상속하고 있다.

뷰	뷰는 CWnd 유도 클래스인 CView 클래스를 상속하여 만들어진다. 뷰는 다큐먼트에 부착되어 다큐먼트와 사용자 사이의 중개 역할을 한다. 뷰는 SDI 프레임 윈도우나 MDI 차일드 프레임 윈도우의 클라이언트 영역을 차지하는 차일드 윈도우이다.
다이얼로그 박스	다이얼로그 박스는 CWnd 유도 클래스인 CDialog 클래스를 상속하여 만들어진다.
폼	다이얼로그 템플릿 리소스를 바탕으로 만들어지는 폼 뷰는 CWnd 유도 클래스인 CFormView, CRecordView, CDaoRecordView 등의 클래스를 상속하여 만들어진다. 일반적인 뷰의 기능에 다이얼로그에서처럼 컨트롤을 배치할 수 있는 기능이 첨가된 것으로 볼 수 있다.
컨트롤	버튼, 콤보 박스 등 윈도우에서 사용되는 모든 공통 컨트롤들도 CWnd 클래스의 유도 클래스들이다.
컨트롤 바	컨트롤을 포함하고 있는 차일드 윈도우로 툴바 상태바 등이 포함된다.

다) 윈도우의 생성

MFC의 마법사는 필요한 대부분의 윈도우들을 자동으로 생성해준다. 하지만 특별한 목적을 위해서는 직접 윈도우를 생성해야 하는 경우가 있다.

***. 윈도우 클래스 등록**

"윈도우 클래스"는 윈도우 프로그래밍에서 윈도우를 생성하기 위해 필요한 특성들을 정의한다. 이는 C++에서의 클래스와 달리 윈도우 생성을 위한 템플릿 또는 모델을 말한다. 전통적인 윈도우 응용 프로그램에서 윈도우 클래스의 등록은 RegisterClass 함수나 RegisterClassEx 함수를 통해 이루어진다. 이들 함수는 윈도우 클래스의 속성을 나타내는 매개변수를 받아서 유일한 윈도우 클래스를 시스템에 등록해 준다.

MFC에서는 기존의 방법을 통해서도 윈도우 클래스를 등록할 수 있지만 대부분의 윈도우 클래스 등록이 자동적으로 이루어진다. MFC는 이미 등록된 여러 윈도우 클래스를 제공하고 있으며 AfxRegisterWndClass 함수를 통해 필요한 클래스를 추가로 등록할 수 있다. 등록된 클래스는 Create 멤버 함수를 통해 윈도우 생성에 이용된다.

LPCTSTR AFXAPI AfxRegisterWndClass(
UINT nClassStyle,
HCURSOR hCursor = 0,
HBRUSH hbrBackground = 0,
HICON hIcon = 0
);

AfxRegisterWndClass 함수는 내부적으로 RegisterClass 함수를 사용하고 있지만 RegisterClass 함수에 비해 매개변수가 훨씬 적다. 윈도우 등록에 필요한 다른 파라미터들은 디폴트 값을
자동적으로 이용한다. 함수에서 반환되는 문자열은 MFC가 생성해주는 클래스의 이름으로 파라미터를 기초로 하여 생성된다. 따라서 동일한 파라미터를 사용하여 AfxRegisterWndClass 함수를 여러 번 호출하면 두 번째 호출 이후로는 처음 등록된 클래스의 이름이 반환된다. 반환된 문자열은 다음번 AfxRegisterWndClass 함수를 호출할 때까지만 유효한 정적 스트링 버퍼에 대한 포인터이므로 등록한 클래스로 여러 윈도우를 생성할 필요가 있다면 이를 저장해두는 것이 좋다.

```
CString strMyClass;
// 커서, 브러시, 아이콘 등을 리소스에서 로드합니다.
try
{
strMyClass = AfxRegisterWndClass(
CS_VREDRAW | CS_HREDRAW,
::LoadCursor(NULL, IDC_ARROW),
(HBRUSH) ::GetStockObject(WHITE_BRUSH),
::LoadIcon(NULL, IDI_APPLICATION));
}
catch (CResourceException* pEx)
{
AfxMessageBox( _T("Couldn't register class! (Already registered?)"));
pEx->Delete();
```

```
}
// 등록된 클래스로 윈도우를 생성합니다.
CWnd* pWnd = new CWnd;
pWnd->Create(strWndClass, ...);
```

AfxRegisterWndClass 함수보다 정교한 등록 작업이 필요하다면 API 함수인 RegisterClass 함수나 MFC 함수인 AfxRegisterClass 함수를 사용하면 된다. DLL에서는 AfxRegisterClass 또는 AfxRegisterWndClass 함수를 사용하는 것이 중요하다. Win32는 DLL이 등록한 클래스를 자동적으로 등록 해제하지 않는다. 따라서 DLL이 종료될 때 명시적으로 등록을 해제해 주어야 한다. AfxRegisterClass 함수는 이러한 등록 해제도 자동적으로 수행해 준다. AfxRegisterClass 함수는 DLL이 등록한 클래스의 목록을 관리하며 DLL이 종료될 때 자동으로 등록 해제한다. RegisterClass 함수를 사용한 경우에는 명시적으로 DllMain 함수 내에서 등록 해제를 해주어야 하며, 등록을 해제하지 않았을 경우 다른 클라이언트가 DLL을 사용하려고 할 때 예기치 않은 오류를 발생시킬 수 있다.

*** 윈도우 생성 순서**

MFC가 제공하는 모든 윈도우 클래스는 2단계 생성 과정을 거친다. 생성 과정에서는 먼저 C++의 new 연산자를 사용하여 C++ 개체를 할당하고 초기화한다. 하지만 이에 해당하는 윈도우는 생성되지 않는다. 실제 윈도우의 생성은 C++ 개체가 할당된 후 Create 멤버 함수를 호출할 때 생성된다. Create 멤버 함수는 윈도우를 생성하고 생성된 윈도우의 HWND를 C++ 개체의 *m_hWnd* 멤버 변수에 저장한. Create 함수를 호출하기 전에 아이콘이나 클래스 스타일 등을 설정하기 위해 AfxRegisterWndClass 함수를 사용하여 윈도우 클래스를 등록할 수 있다. 프레임 윈도우의 경우에는 Create 함수 대신 LoadFrame 함수를 사용한다. LoadFrame 함수는 Create 함수에 비해 필요한 파라미터의 수가 적다. 이는 프레임의 타이틀, 아이콘, 가속키 테이블, 메뉴 등을 리소스에서 얻어오기 때문이다. 이 때 아이콘, 가속키 테이블, 메뉴는 동일한 아이디를 가져야 하며 디폴트 값은 IDR_MAINFRAME이다.

라) 윈도우 개체의 파괴

윈도우가 종료될 때에는 차일드 윈도우 개체의 파괴에 주의하여야 한다. 차일드 윈도우가 파괴되지 않으면 응용 프로그램은 차일드 윈도우를 위한 메모리를 복구할 수 없다. 다행히 MFC는 프레임 윈도우, 뷰, 다이얼로그 박스 등에 대해서 생성뿐만이 아니라 파괴도 관리해주고 있다. 추가적으로 윈도우를 생성한 경우에는 이들을 파괴하는 책임은 프로그래머에게 있다.

*** 윈도우 파괴 순서**

MFC에서 사용자가 프레임 윈도우를 닫으면 WM_CLOSE 메시지가 발생하고 이 메시지의 핸들러인 OnClose 함수가 호출된다. OnClose 함수는 다시 DestroyWindow 함수를 호출한다. 윈도우가 파괴될 때 호출되는 마지막 OnNcDestroy 함수로 윈도우의 파괴를 위해 Default 멤버 변수를 호출한다. 윈도우가 파괴된 이후에는 PostNcDestroy 함수가 호출된다. CFrameWnd 클래스에서 PostNcDestroy 함수의 기본 구현은 C++ 윈도우 개체를 삭제하도록 되어 있다.

한 가지 주의할 점은 프레임 윈도우나 뷰의 경우 윈도우를 파괴하기 위해 C++의 delete 연산자를 사용해서는 안되며 CWnd 클래스의 DestroyWindow 함수를 사용하여야 한다는 점이다. PostNcDestroy 함수에서 윈도우 개체를 삭제하고 있으므로 이를 다시 삭제하고자 한다면 오류를 발생시킬 것이다. 다큐먼트나 응용 프로그램 클래스는 윈도우 클래스가 아니므로 위의 순서에 따르지 않는다.

*** CWnd와 HWND의 분리**

CWnd 개체에서 윈도우 핸들을 분리하기 위해서 MFC는 Detach 함수를 제공하고 있다. 이는 윈도우 개체의 소멸자가 윈도우를 파괴하지 못하도록 하기 위해 사용된다.

마) 윈도우 개체의 사용

CWnd 클래스는 윈도우의 기본적인 동작을 모두 구현하고 있으므로 특별한 경우가 아니

면 기본 구현만으로도 필요한 작업을 할 수 있다. 이에 비해 응용 프로그램에 따라서 모두 다른 기능 즉, 프로그래머가 윈도우 개체를 이용함에 있어서 주로 다루게 되는 것은 다음 두 가지 동작이다.

◦ 윈도우 메시지 처리
◦ 윈도우 그리기

윈도우 메시지를 처리하기 위해서는 C++ 윈도우 클래스에 메시지들을 매핑해 주어야 한다. 이는 MFC가 제공하는 메시지 맵을 통해 처리된다. CWnd 클래스는 메시지를 처리할 수 있는 CCmdTarget 클래스의 유도 클래스이므로 생성될 때 메시지 맵을 가지고 있다. 윈도우의 그리기는 윈도우가 가지고 있는 데이터를 사용자를 위해 표현하는 것으로 이러한 그리기 작업은 뷰 클래스의 OnDraw 함수에서 처리된다. 만약 어떤 윈도우가 뷰의 차일드 윈도우라면 뷰의 그리기 작업 중 일부는 차일드 윈도우의 OnDraw 함수 호출을 통해 차일드 윈도우가 처리하도록 할 수 있다.

그리기 작업을 위해서는 장치 컨텍스트가 필요하다. 장치 컨텍스트는 화면이나 프린터와 같은 장치의 그리기 속성을 나타내는 데이터 구조로, 모든 그리기 작업은 장치 컨텍스트 개체를 통해 이루어진다.

바) 장치 컨텍스트

장치 컨텍스트는 화면이나 프린터와 같은 장치의 그리기 속성을 나타내는 윈도우의 데이터 구조이다. 모든 그리기는 장치 컨텍스트 개체의 함수 호출을 통해 이루어진다. 장치 컨텍스트 클래스는 직선, 도형, 문자열 등을 그리기 위한 API 함수들을 캡슐화하고 있다. 이러한 장치 컨텍스트는 장치를 추상적으로 묘사하고 있기 때문에 장치 독립적인 그리기를 가능하게 해준다. 따라서 동일한 그리기 코드를 화면, 프린터, 메타 피일 등에 사용할 수 있다.

CPaintDC 개체는 BeginPaint 함수를 호출하고 장치 컨텍스트를 통해 그리기를 한 후 EndPaint 함수를 호출하는 과정을 캡슐화하고 있다. CPaintDC 클래스의 생성자는 BeginPaint 함수를 호출하고 소멸자는 EndPaint 함수를 호출한다. 따라서 단순히 개체를

생성하고 그리기 작업을 한 후 개체를 소멸시키는 것으로 그리기 작업은 끝난다. 대표적으로 OnDraw 함수로 전달되는 매개변수가 CPaintDC 개체로 OnPrepareDC 함수를 통해 초기화가 끝난 상태이므로 그리기에만 신경을 쓰면 된다. 그리기 작업을 끝낸 장치 컨텍스트는 OnDraw 함수에서 반환한 후 MFC에 의해 자동으로 소멸된다.

CClientDC 개체는 윈도우의 클라이언트 영역만을 나타내는 장치 컨텍스트이다.
CClientDC 클래스의 생성자는 GetDC 함수를 호출하고 소멸자는 ReleaseDC 함수를 호출한다. CWindowDC 개체는 프레임 윈도우를 포함하는 전체 윈도우를 나타내는 장치 컨텍스트이다.
CMetaFileDC 개체는 메타 파일로의 그리기를 캡슐화한 것입니다. 이는 CPaintDC 개체와는 달리 OnPrepareDC 함수를 호출해주어야 한다.

사) 그래픽 개체

윈도우즈 시스템은 장치 컨텍스트에 그리기를 위해 다양한 도구를 제공한다. 예를 들어 선을 그리기 위한 펜, 내부를 칠하기 윈한 브러시, 문자열을 그리기 위한 폰트 등이 여기에 해당한다. MFC는 그리기 도구들을 그래픽 개체 클래스로 캡슐화하여 제공하고 있다. 아래의 표는 MFC에서 제공하는 클래스와 이에 대응하는 그래픽 장치 인터페이스(GDI, Graphic Device Interface)의 핸들을 나타낸 것이다.

Class	Windows handle type
CPen	HPEN
CBrush	HBRUSH
CFont	HFONT
CBitmap	HBITMAP
CPalette	HPALETTE
CRgn	HRGN

각 그래픽 개체 클래스는 윈도우의 경우와 마찬가지로 그래픽 개체를 생성하는 생성자를 가지고 있고, 생성된 클래스를 초기화하는 생성 함수를 가지고 있다. 또한 MFC 그래픽 개

체 클래스를 그에 대응하는 윈도우의 핸들로 캐스팅할 수 있는 연산자를 가지고 있다. 캐스팅된 핸들은 개체가 분리될 때까지 유효하다. 개체의 Detach 함수는 개체에서 핸들을 분리하기 위해 사용된다. 다음 코드는 CPen 개체를 생성하여 윈도우의 핸들로 변환하는 예이다.

```
CPen myPen;
myPen.CreateSolidPen( PS_COSMETIC, 1, RGB(255,255,0) );
HPEN hMyPen = (HPEN) myPen;
```

장치 컨텍스트에서 그래픽 개체를 생성하기 위해서는 다음 4단계가 일반적으로 사용된다.

1. 스택 프레임에 그래픽 개체를 생성하고 생성된 개체를 CreatePen과 같은 생성 함수를 통해 초기화 한다.
2. 개체를 현재의 장치 컨텍스트로 선택한다. 이 때 이전 그래픽 개체를 저장해 두어야 한다.
3. 현재 그래픽 개체로 그리기 작업이 끝난 경우 이전 그래픽 개체를 장치 컨텍스트로 다시 선택하여 원상태로 복구한다.
4. 실행 범위를 벗어나면 자동으로 스택에 할당된 그래픽 개체는 삭제된다. 만약 그래픽 개체가 계속해서 필요하다면 이를 할당해 놓고 필요한 경우에 장치 컨텍스트로 선택 할 수 있다. 이 경우 더 이상 그래픽 개체가 필요하지 않다면 이를 반드시 삭제하여야 한다.

*** 개체의 1단계 또는 2단계 생성**

펜이나 브러시와 같은 그래픽 개체들을 생성하는 방법은 두 가지가 있다.

- *1*단계 생성: 생성과 초기화를 생성자에서 해준다.
- *2*단계 생성: 생성과 초기화를 별도로 해준다.
  생성자는 개체를 생성하고 초기화는 별도의 함수를 통해 해준다.

일반적으로 1단계 생성에 비해 2단계 생성이 더 안전하다. 1단계 생성에서는 매개변수를 잘못 지정하거나 메모리 할당에 실패할 경우 예외를 발생시킨다. 이러한 문제는 2단계 생성 과정에서 오류 검사를 통해 피할 수 있다. 개체를 파괴하는 것은 두 경우 모두 동일하다.

다음은 두 가지 생성 방법을 간단하게 나타낸 것이다.

```
void CMyView::OnDraw( CDC* pDC )
{
// 1단계 생성
CPen myPen1( PS_DOT, 5, RGB(0,0,0) );
// 2단계 생성 : 펜 개체를 생성합니다.
CPen myPen2;
// 펜을 초기화합니다.
if( myPen2.CreatePen( PS_DOT, 5, RGB(0,0,0) ) )
// 펜 사용
}
```

*** 장치 컨텍스트로 그래픽 개체의 선택**

그래픽 개체를 생성한 후에는 장치 컨텍스트로 생성한 그래픽 개체를 선택해 주어야 한다. 다음은 펜을 생성하여 그리기를 수행하는 간단한 예이다.

```
void CMyView::OnDraw( CDC* pDC )
{
CPen penBlack; // 펜을 생성합니다.
// 펜을 초기화합니다.
if( newPen.CreatePen( PS_SOLID, 2, RGB(0,0,0) ) )
{
// 장치 컨텍스트로 펜을 선택합니다.
```

```
// 이 때 이전 펜을 저장해야 합니다.
CPen* pOldPen = pDC->SelectObject( &penBlack );
// 펜을 이용하여 그림을 그립니다.
pDC->MoveTo(...);
pDC->LineTo(...);
// 저장된 이전 펜을 복구합니다.
pDC->SelectObject( pOldPen );
}
else
{
// 펜 초기화 실패를 알려줍니다.
}
}
```

SelectObject 함수에서 반환한 그래픽 개체는 임시적이다. 즉 다음번에 CWinApp 클래스의 OnIdle 함수가 호출되는 경우 삭제된다. 하지만 SelectObject 함수에서 반환된 개체를 하나의 함수 내에서만 사용하고 사용 도중 메시지 루프로 제어권을 넘기지 않는다면 아무런 문제가 없다.

1.3.3 다큐먼트-뷰 구조

기본적으로 MFC의 응용 프로그램 마법사는 다큐먼트와 뷰 클래스를 포함하는 코드를 만들어 낸다. 다큐먼트/뷰 구조는 데이터 자체를 그 표현 및 사용자의 조작으로부터 분리하고 있다.

다큐먼트는 데이터를 저장하고 데이터를 관리하며 데이터의 변경에 따른 뷰의 업데이트를 조절한다. 또한 다큐먼트는 데이터베이스와 같은 기억 장치에서 데이터를 얻어오기 위해 인터페이스를 제공할 수도 있다.

뷰는 데이터를 표시하고 데이터의 선택, 편집 등 사용자와의 상호작용을 관리한다. 뷰는

표현할 데이터를 다큐먼트로부터 얻어오며 데이터가 변경된 경우 이를 다시 다큐먼트에 알려준다. 다큐먼트/뷰 구조를 이용하지 않을 수도 있지만, 대부분의 응용 프로그램에서 이 구조는 적합하다.

예를 들어 하나의 데이터를 여러 가지 형식으로 표현하고 싶은 경우 하나의 다큐먼트와 여러 개의 뷰를 사용하여 구현할 수 있다. 이는 다큐먼트/뷰 구조가 데이터로부터 그 표현을 분리하고 있기 때문에 가능한 것으로 뷰에 다큐먼트가 포함된 경우라면 데이터의 중복이 발생할 수 있어 비효율적이다.

MFC의 다큐먼트/뷰 구조는 다중 뷰, 다중 다큐먼트 타입, 분할 윈도우 등 다양한 사용자 인터페이스 구조를 가능하게 해준다.

다큐먼트-뷰 구조는 4개의 핵심 클래스로 구성되어 있다.

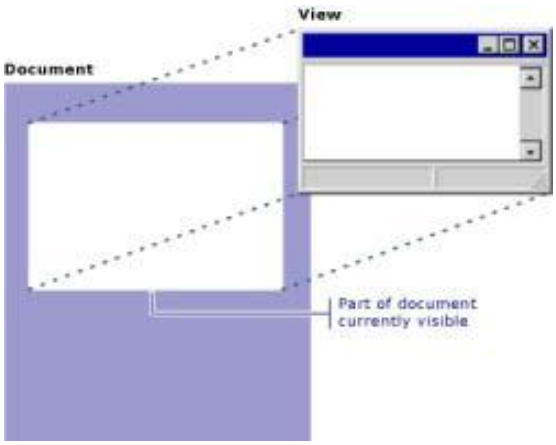
- 다큐먼트 : CDocument 클래스는 데이터를 저장하거나 프로그램에서 사용하는 데이터를 제어하기 위한 기능을 제공한다. 다큐먼트는 파일 메뉴에서 파일 열기 또는 저장 항목과 관련된 데이터 단위를 나타낸다.

- 뷰 : CView 클래스는 뷰와 관련된 기본 기능을 제공합니다. 뷰는 다큐먼트에 연결되어 다큐먼트와 사용자 사이의 중개 역할을 한다. 뷰는 다큐먼트를 화면이나 프린터에 표시하고 사용자의 입력을 다큐먼트로 전달한다.

- 프레임 윈도우 : CFrameWnd 클래스는 다큐먼트를 나타내는 하나 이상의 뷰에 프레임을 제공한다.

- 다큐먼트 템플릿 : CDocTemplete 클래스는 서로 연관되어 있는 다큐먼트, 뷰 그리고 프레임 윈도우 개체들을 생성하고 이들을 관리하는 책임을 진다.

다음 그림은 다큐먼트와 뷰의 관계를 나타낸 것이다.



가) 다큐먼트-뷰 구조 개요

다큐먼트와 뷰는 MFC를 이용한 응용 프로그램에서 항상 쌍으로 존재한다. 데이터는 다큐먼트에 저장되지만 뷰는 이들 데이터를 억세스할 수 있는 권리를 가진다.

다큐먼트/뷰 쌍에서 뷰는 다큐먼트의 데이터를 억세스하기 위해 여러 가지 방법을 사용할 수 있다.

먼저 다큐먼트에 대한 포인터를 반환하는 GetDocument 멤버 함수를 사용할 수 있다. GetDocument 함수는 뷰와 연결되어 있는 다큐먼트의 포인터를 반환하므로 다큐먼트의 멤버 변수를 직접 억세스하거나 멤버 함수를 통해 간접 억세스할 수 있다. 다른 방법은 뷰 클래스를 다큐먼트 클래스의 friend 클래스로 만드는 것이다.

대표적인 예가 데이터를 화면에 표시하기 위한 뷰의 OnDraw 함수로, 뷰는 다큐먼트에 대한 포인터를 얻기 위해 GetDocument 함수를 사용하고 있다.

```
void CMDIView::OnDraw(CDC* /*pDC*/)
{
CMDIDoc* pDoc = GetDocument();
```

```
ASSERT_VALID(pDoc);
// TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
}
```

뷰는 데이터를 선택하거나 편집하기 위해 사용자가 마우스로 뷰의 영역을 클릭한 경우 이를 처리하며, 키보드 입력의 처리도 책임지고 있다. 사용자가 문자열을 표시하고 있는 뷰에 새로운 문자열을 입력한 경우를 생각해 보면, 뷰는 다큐먼트에 대한 포인터를 얻고 입력된 문자열을 다큐먼트로 전달하여 다큐먼트가 데이터를 저장하도록 할 것이다.

분할 윈도우와 같이 동일한 다큐먼트에 여러 뷰가 연결된 경우 뷰는 먼저 새로운 데이터를 다큐먼트로 전달한다. 다음으로는 다큐먼트의 UpdateAllViews 멤버 함수를 호출하여 모든 뷰에게 업데이트하도록 알려준다.

나) 다큐먼트-뷰 구조의 장점

다큐먼트/뷰 구조의 가장 큰 장점은 하나의 다큐먼트에 여러 개의 뷰를 사용할 수 있다는 점이다. 표 형식으로 나타나는 숫자 데이터를 다루는 응용 프로그램을 생각해 보면. 사용자는 이 데이터가 숫자뿐만이 아니라 그래프 형식으로 보기를 원할 수 있다. 이런 경우 분할 윈도우를 사용하여 간단하게 하나의 다큐먼트에 대해 두 가지 형식의 데이터 표현을 보여줄 수 있다. 사용자가 숫자 데이터를 편집하는 경우, 그래프를 나타내는 또 다른 뷰는 사용자가 입력한 데이터를 반영해서 고쳐져야 한다. 이를 위해 뷰는 CDocument::UpdateAllViews 함수를 통해 그래프가 업데이트되도록 할 수 있다.

다른 방법을 통해서 이러한 동작을 구현하는 것이 가능하지만, 만약 뷰에 데이터가 저장되는 경우 이는 데이터의 중복이 불가피하게 될 것이다. 다큐먼트/뷰 구조는 대부분의 응용 프로그램에서 손쉽게 사용할 수 있는 구조라 할 수 있다.

다) 다큐먼트-뷰 구조를 사용하지 않는 경우

MFC는 정보 관리와 데이터 시각적인 표현을 위해 일반적으로 다큐먼트/뷰 구조를 사용한다.

대부분의 응용 프로그램에서 다큐먼트/구조는 적절하고 효율적인 방법이라 할 수 있다. 다큐먼트/뷰 구조는 데이터를 그 표현과 분리하고 있어서 응용 프로그램 개발을 단순화 시키고 불필요한 코드를 줄여준다. 하지만 모든 경우에 다큐먼트/구조가 적합한 것은 아니다. 다음 예들을 살펴보자.

- C 언어로 작성한 윈도우 프로그램을 변환하는 경우에는 다큐먼트/뷰 구조를 사용하지 않고 변환하기를 원할 수 있다. 이런 경우에는 다큐먼트/뷰 구조를 사용하지 않는 것이 변환 과정을 간단하게 할 수 있다.

- 간단한 유틸리티와 같이 단순한 작업을 위한 프로그램을 작성하는 경우에는 다큐먼트/뷰 구조 없이도 가능하며, 다큐먼트/뷰 구조를 사용하는 것이 오히려 부담이 될 수 있다.

- 원본 코드에서 데이터 자체와 데이터 표현을 위한 코드들이 섞여있는 경우 이를 분리하여 다큐먼트/뷰 구조로 만드는 것은 힘든 작업이 될 수 있다. 이런 경우 원본 코드를 그대로 사용할 수 있다.

다큐먼트/뷰 구조를 사용하지 않는 응용 프로그램을 생성하기 위해서는 응용 프로그램 마법사에서 "Document/View architecture support" 옵션을 제거하면 된다.

다이얼로그 기반 응용 프로그램은 다큐먼트/뷰 구조를 사용하지 않으므로 이 형식을 선택한 경우에 다큐먼트/뷰 구조 지원 체크 박스는 디스에이블 상태가 된다.

다큐먼트/뷰 구조를 사용하지 않더라도 툴바, 스크롤바, 상태바 등을 사용할 수 있다. 하지만 다큐먼트/뷰 구조를 사용하지 않으면 다큐먼트 템플릿을 등록하지 않으며 다큐먼트 클래스도 생성되지 않는다.

다큐먼트/뷰 구조는 응용 프로그램의 기본적인 기능들을 구현해주고 있다. 만약 다큐먼트/뷰 구조를 사용하지 않는다면 다큐먼트/뷰 구조에서 지원되는 다음 기능들은 직접 구현하여야 한다.

◦ 응용 프로그램의 파일 메뉴에 "새 파일"과 "종료"만이 제공된다. "새 파일"의 경우에도 MDI 형식에서만 지원되고 SDI 형식에서는 지원되지 않는다. 또한 최근 사용한 파일 목록은 지원되지 않는다.

◦ 파일 메뉴의 "열기"와 "저장"을 포함하여 응용 프로그램이 제공하는 명령에 대한 핸들러 함수를 직접 구현하여야 한다. MFC는 이들 명령에 대한 기본 구현을 제공하지 않는다.

◦ 제공되는 툴바에는 최소한의 버튼만이 제공된다.

라) 다큐먼트 클래스의 이용

다큐먼트와 뷰는 함께 사용되어 다음과 같은 일들을 처리한다.

◦ 응용 프로그램의 데이터들을 저장, 관리, 표현한다.
◦ 데이터 조작을 위한 인터페이스를 제공한다.
◦ 파일로 쓰기와 읽기에 관여한다.
◦ 인쇄하기와 인쇄 미리보기에 관여한다.
◦ 대부분의 명령과 메시지를 처리한다.

다큐먼트는 특히 데이터 관리와 연관이 있다. 데이터는 일반적으로 다큐먼트 클래스의 멤버 변수로 저장되고, 뷰는 이들 변수를 억세스하여 데이터를 표현하고 갱신한다. 파일로 읽기와 쓰기를 위해서 다큐먼트 클래스는 직렬화(serialization)를 제공하고 있다. 이는 CObject 클래스에서 제공하는 기능으로 다큐먼트 개체의 현재 상태를 저장하기 위해 사용된다. 또한 다큐먼트는 명령 메시지를 처리할 수 있다. 하지만 GUI를 갖지 않는 클래스이기 때문에 WM_COMMAND 메시지를 제외한 윈도우 메시지는 처리할 수 없다. 데이터를 조작하기 위한 마우스나 키보드 메시지들은 뷰에서 처리되는 것이 일반적이다.

*** 다큐먼트 클래스의 유도**

다큐먼트 클래스는 응용 프로그램의 데이터를 저장하고 있고 이를 관리한다. 다큐먼트 클

래스를 이용하기 위해서는 일반적으로 다음 작업들이 필요하다.

- CDocument 클래스의 유도 클래스를 생성합니다. 이는 마법사가 생성해 준다.
- 데이터 저장을 위한 멤버 변수를 추가한다.
- CDocument 클래스의 Serialize 멤버 함수를 오버라이드한다. Serialize 함수는 다큐먼트의 데이터를 파일로 읽거나 쓰기 위해 사용된다.

이외에도 많이 오버라이드할 수 있는 함수는 많다. 특히 OnNewDocument 함수와 OnOpenDocument 함수는 다큐먼트의 데이터를 초기화하기 위해 오버라이드할 수 있다. DeleteContents 함수는 동적으로 할당된 데이터를 삭제하기 위해 오버라이드한다.

*** 데이터 관리**

다큐먼트의 데이터는 다큐먼트 클래스의 멤버 변수로 구현된다. 멤버 변수의 구현은 프로그래머가 응용 프로그램의 필요에 따라 구현하는 것으로 마법사가 생성하는 다큐먼트 클래스에는 데이터 저장을 위한 변수가 없다.

데이터 저장을 위해서는 MFC가 제공하는 collection class인 배열, 리스트, 맵 등의 클래스를 활용할 수 있으며, 흔히 사용되는 데이터들을 위해서는 CString, CRect, CPoint, CSize, CTime 등의 클래스를 사용할 수 있다.

뷰는 다큐먼트 개체의 포인터를 통해 다큐먼트의 데이터를 억세스한다. 다큐먼트와 뷰는 생성될 때 CDocTemplate 클래스에 의해 연결되어 있으므로 다큐먼트의 포인터를 얻기 위해서는 CView 클래스의 GetDocument 멤버 함수를 사용하면 된다. 이 때 GetDocument가 반환한 포인터는 응용 프로그램의 다큐먼트 클래스로 캐스팅하여 사용하여야 한다.

*** 데이터의 직렬화 (serialization)**

영속성(persistence)에 대한 기본 개념은 개체를 현재의 상태 그대로 저장장치에 저장하

였다가 이후 개체를 다시 생성할 때 이 데이터를 읽어 들여 원래의 상태로 복구하는 것을 말한다. 여기에서 중요한 점은 개체 자신이 자신의 상태를 읽거나 쓰는데 책임이 있다는 점이다. 따라서 영속적인 클래스의 경우 직렬화를 구현하여야 한다.

MFC는 다큐먼트를 저장하는 "Save"나 "Save As" 명령과 다큐먼트를 읽어오는 "Open" 명령에 대한 기본 구현을 제공하고 있으므로 적은 노력으로 데이터를 읽고 쓸 수 있다. 이를 위해 반드시 오버라이드하여야 하는 멤버 함수는 다큐먼트 클래스의 Serialize 함수이다.

MFC의 응용 프로그램 마법사는 CDocument 유도 클래스에 Serialize 함수를 오버라이드하여 작성해준다. 데이터 저장을 위한 멤버 변수들을 정의한 후에 Serialize 함수를 구현함으로써 다큐먼트의 데이터들을 파일과 연결된 저장 개체(archive object)로 보낼 수 있다. CArchive 개체는 표준 C++에서의 입출력 스트림에서 제공하는 cin이나 cout과 유사하지만 텍스트 형식이 아닌 이진 형식으로 기록하는 점이 다르다.

마법사가 생성한 Serialize 함수는 파일로 읽기와 쓰기를 구별해 주는 코드만을 생성해준다. 실제 데이터의 전송은 다큐먼트의 데이터에 따라 프로그래머가 작성하여야 한다.

```
void CMDIDoc::Serialize(CArchive& ar)
{
if (ar.IsStoring())
{
// TODO: 여기에 저장 코드를 추가합니다. 파일로 쓰기
}
else
{
// TODO: 여기에 로딩 코드를 추가합니다. 파일에서 읽기
}
}
```

◎ 직렬화의 기본 개념

직렬화는 개체를 영구적인 저장 장치로 읽거나 쓰는 과정을 말한다. 직렬화는 MFC에서 CObject 클래스를 통해 지원되므로 CObject 클래스에서 상속된 모든 클래스는 직렬화를 사용할 수 있다.

CObject 클래스는 MFC에서 제공하는 모든 클래스들의 최상위 클래스이므로 MFC의 모든 클래스들은 직렬화를 사용할 수 있다. 직렬화의 기본 개념은 개체가 멤버 변수의 값에 의해 현재 상태로 저장될 수 있고 이후 이들 멤버 변수의 값들을 얻어옴으로써 저장 당시의 상태로 복구할 수 있다는 것이다.

MFC는 CArchive 개체를 사용하여 직렬화될 개체와 저장 장치 사이를 연결한다. CArchive 개체는 일반적으로 CFile 개체와 연결되어 있어서 파일로부터 직렬화에 필요한 파일 이름이나 읽기/쓰기 상태 등의 정보를 얻어온다. 직렬화를 수행하는 개체는 저장 장치의 특성과 상관없이 CArchive 개체를 사용할 수 있다.

CArchive 개체는 쓰기와 읽기 동작을 위해 삽입(<<)과 추출(>>) 연산자를 오버라이드하고 있다. 한 가지 주의할 점은 CArchive 클래스가 표준 입출력 스트림을 위한 클래스와 유사하지만 형식화된 문자열을 지원하는 표준 입출력 스트림 클래스와 달리 CArchive 클래스는 이진 데이터를 입출력한다는 점이다.
원하는 경우 MFC가 제공하는 직렬화를 사용하지 않을 수도 있다. 이 경우에는 직렬화 함수를 호출하는 명령들을 오버라이드하여야 한다. ID_FILE_OPEN, ID_FILE_SAVE, 그리고 ID_FILE_SAVE_AS 표준 명령이 직렬화를 사용하고 있다.

◎ 직렬화에서 다큐먼트의 역할

MFC는 파일 메뉴의 열기, 저장, 다른 이름으로 저장 명령에 대해 자동적으로 다큐먼트의 Serialize 함수를 호출한다. ID_FILE_OPEN 명령의 경우 응용 프로그램 개체의 핸들러 함수를 호출한다. 여기에서 사용자는 파일 열기 다이얼로그 박스에서 파일 이름을 선택하고, MFC는 CArchive 개체를 생성하여 다큐먼트 클래스의 Serialize 함수를 호출하게 된다.

MFC가 이미 파일을 열어 놓았기 때문에 Serialize 함수에서는 데이터를 읽어서 필요한 경우 이를 재구성하기만하면 된다.

◎ 직렬화에서 데이터의 역할

일반적으로 클래스 형식의 데이터는 자신을 직렬화할 수 있어야 한다. 즉 한 개체가 저장 개체로 전달된 경우 전달된 개체는 저장 개체로 자신을 쓰는 방법과 저장 개체로부터 읽는 방법을 알고 있어야 한다. MFC가 제공하는 데이터 형식의 클래스들은 직렬화 함수를 구현하고 있으므로 MFC가 제공하는 데이터 형식의 멤버 변수들은 CArchive 개체로 전달해주기만 하면 직렬화가 수행된다.

새로운 데이터 형식의 클래스를 작성하는 경우에는 직렬화를 구현해주어야 한다.

◎ MFC의 직렬화 사용하지 않기

MFC는 파일로의 읽기와 쓰기를 위한 방법으로 직렬화를 제공하고 있다. 저장 개체를 통한 직렬화는 대부분의 응용 프로그램에 적합하다. 직렬화를 이용하면 파일 전체를 메모리로 읽어 들이고 사용자가 메모리의 내용을 수정한 후 다시 파일로 모든 데이터를 기록할 수 있다.

하지만 모든 응용 프로그램이 이와 같은 방법으로 데이터를 다루지 않으므로 직렬화가 적합하지 않은 경우도 있다. 예를 들면 데이터베이스 프로그램, 아주 큰 파일의 일부만을 편집하는 프로그램, 텍스트 형식의 출력을 하는 프로그램, 데이터 파일을 공유하는 프로그램 등은 직렬화를 사용할 수 없다.

이런 경우에는 Serialize 함수를 오버라이드하여 CArchive 개체와 작업하는 것이 아니라 CFile 개체와 직접 작업하도록 해야 한다. 이 때에는 CFile 클래스의 Open, Read, Write, Close, Seek 등의 멤버 함수를 통해 CFile 개체와 직접 작업할 수 있다. MFC는 선택된 파일을 열어 놓으므로 CArchive 클래스의 GetFile 멤버 함수를 사용하면 CFile 개체의 포인터를 얻을 수 있다.

Serialize 함수를 완전히 배제하고 싶다면 CWinApp 클래스의 OnOpenDocument 함수와

OnSaveDocument 함수를 오버라이드하면 된다.

*** 다큐먼트에서 명령 메시지의 처리**

다큐먼트는 메뉴 항목, 툴바 버튼, 가속키 등에서 발생하는 명령을 처리할 수 있다. 기본적으로 CDocument 클래스는 직렬화를 이용하여 "저장"과 "다른 이름으로 저장" 명령을 처리한다. 이외에도 다큐먼트의 데이터에 영향을 미치는 명령들을 처리할 것입니다. 다큐먼트는 뷰와 마찬가지로 메시지 맵을 가지지만 GUI를 가지지 않으므로 표준 윈도우 메시지는 처리하지 않는다. 단지 WM_COMMAND 메시지만을 처리한다.

마) 뷰 클래스의 이용

뷰는 다큐먼트의 데이터를 표현하고 사용자로부터 입력을 받아서 이를 해석하고 다큐먼트에 전달하는 역할을 한다. 뷰 클래스에 프로그래머는 일반적으로 다음 코드들을 작성하여야 한다.

- 다큐먼트 데이터를 표현하기 위한 OnDraw 함수 구현
- 데이터와 관련된 윈도우 메시지와 명령들을 핸들러 함수로 연결
- 사용자 입력을 처리하는 핸들러 함수의 구현

추가적으로 CView 클래스의 함수를 오버라이드할 필요가 있을 수 있다. 특히 뷰 클래스의 초기화를 위해서 OnInitialUpdate 함수를 오버라이드할 수 있으며, 뷰가 화면을 업데이트하기 직전에 필요한 처리를 위해 OnUpdate 함수를 오버라이드할 수 있다. 여러 페이지의 다큐먼트를 사용한다면 인쇄를 할 때 페이지 수를 설정하기 위해 OnPreparePrinting 함수를 오버라이드하면 된다.

*** CView 유도 클래스**

CView 클래스와 그 유도 클래스는 프레임 윈도우의 클라이언트 영역을 나타내는 차일드 윈도우이다. 뷰는 다큐먼트의 데이터를 보여주고 다큐먼트를 위해 사용자로부터 입력을 받는다. 뷰 클래스는 생성될 때 다큐먼트 템플릿 개체를 통해 다큐먼트 클래스 및 프레임

윈도우 클래스와 연결되어 있다.
CView 클래스는 뷰 클래스의 기본 기능을 제공한다. 뷰는 다큐먼트에 부착되어 다큐먼트와 사용자의 매개 역할을 한다. 뷰는 다큐먼트의 내용을 화면이나 프린터로 출력하고 사용자의 입력을 다큐먼트로 해석해서 전달한다.

뷰는 프레임 윈도우의 차일드 윈도우이다. 분할 윈도우의 경우처럼 하나 이상의 뷰가 프레임 윈도우를 공유할 수 있다. 뷰 클래스, 프레임 윈도우 클래스, 다큐먼트 클래스는 CDocTemplate 개체에 의해 상호 연결된다. 사용자가 새로운 윈도우를 열거나 기존 윈도우를 나누는 경우 MFC는 새로운 뷰를 만들어서 다큐먼트에 추가한다.

뷰는 단 하나의 다큐먼트에만 연결될 수 있지만 다큐먼트는 동시에 여러 개의 뷰에 연결될 수 있다. 또한 응용 프로그램은 한 종류의 다큐먼트에 여러 종류의 뷰를 제공할 수 있다. 이들 여러 가지 뷰들은 분할 윈도우에 동시에 보여질 수 있다.

뷰 클래스는 메뉴, 툴바, 스크롤바 등의 명령과 더불어 키보드 입력, 마우스 입력, 끌어 놓기에 의한 입력 등 몇 가지 종류의 입력을 처리할 책임을 가지고 있다. 뷰 클래스는 프레임 윈도우에서 전해진 명령을 받는다. 만약 뷰가 전달된 명령을 처리하지 않으면 다큐먼트로 그 처리 기회가 넘어간다. 다른 command target과 마찬가지로 뷰 클래스도 메시지 맵을 통해 메시지를 처리한다.

뷰는 다큐먼트의 데이터를 보여주고 수정하는 책임을 가지고 있지만 데이터를 저장하지는 않는다. 다큐먼트는 뷰에 필요한 데이터를 제공한다. 뷰는 다큐먼트의 데이터를 직접 또는 멤버 함수를 통해 억세스할 수 있다.

다큐먼트에 저장된 데이터가 변경된 경우 뷰는 다큐먼트의 CDocument::UpdateAllViews 함수를 호출한다. CDocument::UpdateAllViews 함수는 이 함수를 호출한 뷰를 제외한 다른 뷰들의 OnUpdate 멤버 함수를 호출하여 변경된 데이터를 반영하도록 한다.
OnUpdate 함수의 기본 구현은 클라이언트 영역 전체를 무효화시키도록 되어 있다.

CView 클래스를 사용하기 위해서는 먼저 CView 클래스의 유도 클래스를 생성하고 OnDraw 함수에 화면 출력을 위한 코드를 작성한다. OnDraw 함수는 인쇄와 인쇄 미리보

기를 위해서도 사용한다.

뷰는 CWnd::OnHScroll 함수와 CWnd::OnVScroll 멤버 함수를 통해 스크롤바 메시지를 처리한다. 스크롤바 메시지를 처리하기 위해서는 이 함수들을 이용하거나 CView 클래스의 유도 클래스인 CScrollView 클래스를 사용하면 된다.

CScrollView 클래스 이외에도 MFC는 여러 종류의 CView 유도 클래스들을 제공하고 있다.

- CView : 다큐먼트의 데이터를 표시하기 위한 뷰의 기본 클래스이다.
- CScrollView : 스크롤 기능을 가진 뷰 클래스입니다. 이 클래스의 유도 클래스를 사용하면 자동으로 스크롤 기능을 가진다.

◎ 폼 뷰와 레코드 뷰

폼 뷰는 스크롤 뷰의 일종으로 다이얼로그 박스 템플릿을 기초로하여 만들어 진다. 레코드 뷰는 폼 뷰의 유도 클래스로 다이얼로그 박스 템플릿과 더불어서 데이터베이스에 대한 연결 기능을 가지고 있다.

◦ CFormView : 레이아웃이 다이얼로그 박스 템플릿으로 정의된 스크롤 뷰이다. 다이얼로 그 박스 템플릿에 기초하여 사용자 인터페이스를 구현하고자 할 때 CFormView 클래스 를 상속하면 된다.

◦ CRecordView : ODBC 레코드 집합을 나타내는 개체에 대한 연결을 가진 폼 뷰이다.
◦ CDaoRecordView : 데이터 억세스 개체(DAO) 레코드 집합을 나타내는 개체에 대한 연결을 가진 폼 뷰이다.
◦ COLEDBRecordView : 폼 뷰에 OLE DB에 대한 지원을 추가한 뷰이다.
◦ CHtmlView : 웹 브라우징을 지원하는 컨트롤을 포함하는 폼 뷰이다. 웹 브라우저 컨트롤은 다이나믹 HTML을 지원한다.
◦ CHtmlEditView : HTML 문서의 편집 기능을 지원하는 폼뷰이다.

◎ 컨트롤 뷰

컨트롤 뷰는 특정 컨트롤을 뷰로 표시한다.
◦ CCtrlView : 모든 컨트롤 뷰의 베이스 클래스이다.
◦ CEditView : 표준 에디트 컨트롤을 뷰로 가진다.
◦ CRichEditView : 리치 에디트 컨트롤을 뷰로 가진다.
◦ CListView : 리스트 컨트롤을 뷰로 가진다.
◦ CTreeView : 트리 컨트롤을 뷰로 가진다.

이외에도 CView 클래스의 유도 클래스 중에는 인쇄 미리보기를 위해 사용하는 CPreviewView 클래스가 있다. 이 클래스는 인쇄 미리보기 윈도우에만 사용되는 툴바, 한 페이지 또는 두 페이지 보기, 확대/축소 등의 기능을 제공한다. CPreviewView 클래스는 일반적으로 그대로 사용하며 유도 클래스를 만드는 경우는 거의 없다. MSDN에도 CPreviewView 클래스는 문서화되어 있지 않다.

*** 뷰에 그리기**

응용 프로그램은 거의 모든 데이터 표현 즉, 그리기 작업은 뷰의 OnDraw 함수에서 일어난다. OnDraw 함수는 기본 구현이 없으므로 반드시 작성해야만 한다. OnDraw 함수는 다

음 작업을 수행한다.

1. 다큐먼트의 멤버 변수에서 데이터를 얻어온다.
2. MFC가 OnDraw 함수로 전달하는 장치 컨텍스트 개체를 이용하여 다큐먼트 데이터의 그리기를 수행한다.

다큐먼트의 데이터가 수정되면 뷰는 변화를 반영하기 위해 다시 그려져야 한다. 일반적으로 이러한 작업은 뷰를 통해 사용자가 다큐먼트의 데이터를 수정한 경우 일어난다. 이 때 뷰는 다큐먼트의 UpdateAllViews 함수를 호출하여 동일한 다큐먼트와 연결되어 있는 모든 뷰들이 데이터를 업데이트 하도록 알려준다. UpdateAllViews 함수는 각 뷰의 OnUpdate 함수를 호출한다. OnUpdate 함수의 기본 구현은 전체 클라이언트 영역을 무효화시킨다. 다큐먼트에서 수정된 부분에 해당하는 뷰의 클라이언트 영역만을 무효화시키기 위해서는 OnUpdate 함수를 오버라이드하면 된다.

뷰가 무효화되면 윈도우는 WM_PAINT 메시지를 보낸다. 메시지를 받은 뷰의 OnPaint 함수는 CPaintDC 클래스의 장치 컨텍스트 개체를 생성하고 뷰의 OnDraw 함수를 호출한다. 일반적으로 OnPaint 함수는 오버라이드할 필요가 없다.

장치 컨텍스트는 화면이나 프린터와 같은 장치의 그리기 속성을 나타내는 데이터 구조로 모든 그리기 작업은 장치 컨텍스트 개체를 통해 이루어진다. 화면에 그리기 작업을 하기 위해서는 OnDraw 함수에 CPaintDC 개체가 전해지고, 프린터를 위해서는 CDC 개체가 전해진다.

일반적으로 뷰 클래스의 OnDraw 함수는 먼저 다큐먼트에 대한 포인터를 얻고, 매개변수로 전달된 장치 컨텍스트를 사용하여 그리기 작업을 수행한다. 다음은 OnDraw 함수 구현의 예이다.

```
void CMyView::OnDraw( CDC* pDC )
{
CMyDoc* pDoc = GetDocument(); // 다큐먼트에 대한 포인터 얻기
CString s = pDoc->GetData(); // 다큐먼트의 데이터 얻기
```

```
CRect rect;
GetClientRect( &rect );
pDC->SetTextAlign( TA_BASELINE | TA_CENTER );
pDC->TextOut( rect.right / 2, rect.bottom / 2,
s, s.GetLength() ); // 그리기 작업
}
```

위의 예는 다큐먼트에서 얻은 문자열을 뷰의 중앙에 출력한다. 화면 출력의 경우 전달되는 파라미터 *pDC*는 CPaintDC 클래스에 대한 포인터로 생성될 때 BeginPaint 함수를 호출하므로 따로 호출할 필요가 없다. 또한 장치 컨텍스트가 파괴될 때 EndPaint 함수를 자동으로 호출해 준다.

*** 사용자 입력**

뷰의 다른 중요한 임무는 사용자의 입력을 받아서 처리하는 것이다. 뷰는 일반적으로 다음 메시지들을 처리한다.

- 마우스와 키보드에 의해 발생되는 윈도우 메시지
- 메뉴, 툴바 버튼, 가속키에서 발생하는 명령 메시지

어느 메시지를 처리할 것인지는 응용 프로그램의 필요에 따라 달라진다. 응용 프로그램이 뷰 내부에서 마우스의 왼쪽 버튼을 눌러 직선을 그리는 경우를 생각해 보자. 이를 위해서 뷰는 WM_LBUTTONDOWN, WM_MOUSEMOVE, 그리고 WM_LBUTTONUP 메시지를 처리할 것이다. 또한 뷰에서는 다큐먼트의 데이터와 관련된 편집 메뉴의 명령들도 처리한다. 복사, 잘라내기, 붙여넣기 등의 명령은 CWnd 클래스의 클립보드 관련 멤버 함수들을 통해 처리된다.

*** 인쇄**

뷰는 화면 출력뿐만이 아니라 인쇄도 처리한다. 이는 윈도우즈 시스템의 장치 컨텍스트가 장치 독립적이기 때문에 가능한 것이다. 뷰가 인쇄 작업에서 하는 일은 다음과 같은 것들이다.

◦ 뷰 클래스는 화면 출력뿐만이 아니라 인쇄를 위해서도 OnDraw 함수를 사용한다.
◦ 여러 페이지로 구성되는 다큐먼트를 인쇄를 위해 페이지 단위로 분할한다.

*** 스크롤**

MFC는 스크롤되는 뷰와 프레임 윈도우의 크기에 자동으로 맞추어지는 뷰를 지원한다. CScrollView 클래스는 두 종류의 스크롤을 모두 지원하는 CView 유도 클래스이다.

◎ 뷰 스크롤하기

다큐먼트의 크기가 뷰가 한 번에 보여줄 수 있는 크기보다 큰 경우는 허다하다. 이는 다큐먼트 데이터의 크기가 증가하거나 프레임 윈도우의 크기가 줄어들어서 발생할 수도 있다. 이런경우 뷰는 원하는 다큐먼트의 데이터를 보여주기 위해 스크롤이 필요하게 된다. 모든 뷰는 OnHScroll 함수와 OnVScroll 함수를 통해 스크롤바 메시지를 처리할 수 있다. CView의 유도 클래스를 사용하는 경우 이들 멤버 함수를 통해 직접 구현함으로써 스크롤 기능을 추가할 수 있지만, CScrollView의 유도 클래스를 사용하는 경우에는 추가적인 코드 작성이 없이도 뷰가 제공하는 스크롤 기능을 사용할 수 있다.

스크롤 뷰에서 스크롤 되는 페이지(스크롤 박스가 이외의 스크롤바를 클릭했을 경우 스크롤 되는 양)와 줄(스크롤 화살표를 클릭했을 경우 스크롤 되는 양)의 크기를 조절하는 것이 가능하다. 이 값은 응용 프로그램에 따라 다르다. 예를 들어 그래픽 프로그램의 경우 줄의 크기는 픽셀 단위일 수 있지만 문서 편집 프로그램의 경우에는 글자의 높이가 되는 것이 일반적이다.

◎ 뷰 크기 바꾸기

뷰가 프레임 윈도우의 크기에 맞게 크기가 조절되기를 원하는 경우에도 CScrollView 클래스를 사용할 수 있다. 논리적인 뷰는 프레임 윈도우의 클라이언트 영역에 맞게 크기가 줄거나 늘어난다. 크기 조절이 가능한 뷰는 스크롤바를 가지지 않는다.

바) 다중 다큐먼트, 뷰

대부분의 응용 프로그램은 한 종류의 다큐먼트만을 지원한다. 하지만 MDI 형식의 응용 프로그램을 사용함으로써 동일한 종류의 다큐먼트를 여러 개 열 수 있다. 이러한 기본 구조에 비해 여러 종류의 다큐먼트나 뷰를 하나의 응용 프로그램에서 사용할 수도 있다.

*** 다중 다큐먼트**

MFC 응용 프로그램 마법사는 하나의 다큐먼트 클래스를 생성해 준다. 하지만 몇몇 경우에는 하나 이상의 다큐먼트 클래스가 필요할 수 있다. 각 다큐먼트는 각각의 다큐먼트 클래스에 의해 표시되며 그와 연결된 서로 다른 뷰를 가질 것이다. 사용자가 "새 파일" 메뉴 항목을 선택하면, MFC는 응용 프로그램이 지원하는 다큐먼트의 종류를 다이얼로그 박스를 통해 보여주고 그 중 사용자가 선택한 종류의 다큐먼트를 생성할 것이다. 각 다큐먼트는 다큐먼트 템플릿 개체에 의해 관리된다.

또 다른 다큐먼트 클래스는 마법사를 통해 CDocument의 유도 클래스를 생성함으로써 추가할 수 있다. 클래스가 생성되면 응용 프로그램 클래스의 InitInstance 함수에서 AddDocTemplate 함수를 한 번 더 호출하여 추가된 다큐먼트 클래스를 등록해주어야 한다.

*** 다중 뷰**

다큐먼트 클래스는 일반적으로 하나의 뷰 만을 사용하지만 하나 이상의 뷰를 하나의 다큐먼트를 위해 사용할 수 있다. 다중 뷰를 위해 다큐먼트는 다큐먼트와 연결된 뷰의 리스트를 관리하며 뷰를 추가하거나 삭제할 수 있는 멤버 함수를 제공한다. 또한 다큐먼트의 데이터가 변경되었을 때 이를 연결된 뷰들에 알려주기 위해 UpdateAllViews 함수를 제공하고 있다.

MFC는 동일한 다큐먼트에 대해 다중 뷰를 지원하는 3가지 방법을 제공하고 있다.

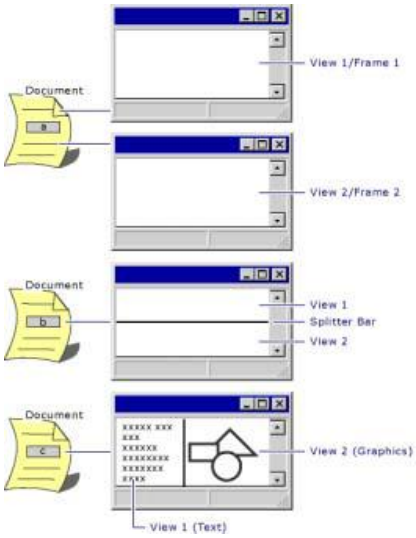
1. 동일한 클래스의 여러 뷰 개체들이 서로 다른 MDI 다큐먼트 프레임 윈도우에 연결되는 경우 하나의 다큐먼트에 대해 두 번째 프레임 윈도우를 생성하는 방법이다. 사

용자가 " 창" 메뉴의 "새 창" 항목을 선택하면 새로운 뷰가 동일한 다큐먼트를 바탕으로 만들어지며, 다큐먼트의 서로 다른 부분을 동시에 살펴보기 위해 사용할 수 있다. MFC는 "새 창" 메뉴 항목을 MDI 형식의 응용 프로그램에서 프레임 윈도우와 뷰를 복사하기 위해 사용 한다. 이 경우 차일드 프레임 윈도우의 타이틀은 "*:1", "*:2 "와 같이 번호가 붙어서 나타 난다.

2. 동일한 클래스의 여러 뷰 개체들이 하나의 다큐먼트 프레임 윈도우에 연결되는 경우 분할 윈도우는 뷰 영역을 여러 영역으로 나누어준다. 이는 동일한 뷰 클래스의 여러 개체를 하나의 다큐먼트에 대해 생성하는 것으로 위의 경우와 프레임 윈도우가 하나라는 것을 제외하고는 동일하다.

3. 서로 다른 클래스의 여러 뷰 개체들이 하나의 다큐먼트 프레임 윈도우에 연결되는 경우 분할 윈도우를 사용하는 것은 두 번째 경우와 같지만 각 뷰는 서로 다른 개체를 바탕으로 생성된다. 이는 동일한 다큐먼트를 서로 다른 형식으로 나타내기 위해 사용된다.

다음 그림은 위에서 설명한 3가지 방법을 나타낸 것이다.



*** 분할 윈도우**

분할 윈도우는 하나 이상의 뷰를 하나의 프레임 윈도우에 보여주기 위해 사용한다. 스크롤바의 분할 컨트롤(또는 분할 박스라고도 불립니다.)을 사용하면 뷰를 분리할 수 있고 분할된 뷰 영역의 넓이를 수정할 수도 있다.

동적인 분할 윈도우에서 각 뷰는 위의 2번째 경우에서처럼 동일한 클래스를 사용하여 뷰가
만들어진다. 정적 분할 윈도우는 위의 3번째 경우에서처럼 다른 클래스를 사용하여 만들어질 수 있다.

사) 다큐먼트와 뷰의 초기화 및 해제

다큐먼트와 뷰를 초기화하고 사용이 끝난 후 해제하기 위해서 다음 내용에 유의하여야 한다.

- MFC는 다큐먼트와 뷰를 초기화한다.
  하지만 다큐먼트에 추가한 멤버 변수들에 대한 초기화는 프로그래머의 몫이다.
- MFC는 사용이 끝난 다큐먼트와 뷰를 자동으로 해제한다. 하지만 다큐먼트와 뷰의 멤버 함수에서 힙에 할당한 메모리를 해제하는 것은 프로그래머의 몫이다.

전체 응용 프로그램에서 필요한 초기화 및 해제를 위해 적합한 함수는 응용 프로그램 클래스의 InitInstance와 ExitInstance 함수이다.

MDI 형식의 응용 프로그램에서 다큐먼트 및 그와 관련된 프레임 윈도우와 뷰의 생성 및 파괴 주기는 다음과 같다.

1. 동적 생성 과정에서 다큐먼트의 생성자가 호출된다.
2. 새로운 다큐먼트에 대해서 OnNewDocument 함수나 OnOpenDocument 함수가 호출된다.
3. 사용자는 다큐먼트와 뷰를 통해 상호 작용한다. 뷰는 다큐먼트에 데이터의 변화를 알려 주고 다큐먼트는 뷰에 업데이트를 알려준다.
4. MFC는 DeleteContents 함수를 호출하여 다큐먼트와 관련된 데이터를 삭제한다.
5. 다큐먼트의 소멸자가 호출된다.

SDI 형식의 응용 프로그램에서는 다큐먼트가 처음 생성될 경우 1번 과정이 한 번만 실행되고 2번에서 4번의 과정이 반복된다. 5번 과정은 응용 프로그램이 종료할 때 한 번만 실행된다.

*** 다큐먼트와 뷰의 초기화**

다큐먼트는 2가지 방법을 통해 생성된다. 따라서 다큐먼트 클래스는 2가지 방법을 모두 지원해야 한다. 먼저 사용자는 "새 파일" 메뉴 항목을 통해 빈 다큐먼트를 생성할 수 있다. 이 경우 다큐먼트의 초기화는 OnNewDocument 함수에서 시행된다.

두 번째는 사용자가 "파일 열기" 메뉴 항목을 통해 다큐먼트를 생성하고 파일로부터 내용을 읽어 들인다. 이 경우 다큐먼트의 초기화는 OnOpenDocument 함수에서 시행된다. 두 경우의 초기화가 동일하다면 두 함수가 공통으로 호출하는 함수를 작성하여 사용할 수도 있고 OnOpenDocument 함수에서 OnNewDocument 함수를 호출하여 사용할 수도 있다.

뷰는 다큐먼트가 생성된 이후에 생성된다. 뷰를 초기화하기에 가장 좋은 시점은 다큐먼트, 프레임 윈도우, 뷰가 모두 생성된 이후이다. 뷰의 초기화는 CView 클래스의 OnInitialUpdate 함수를 통해 이루어진다. 뷰를 다시 초기화하거나 다큐먼트의 변화에 따라 초기화가 바뀌는 경우에는 OnUpdate 함수를 사용하면 된다.

*** 다큐먼트와 뷰의 해제**

다큐먼트를 닫을 때 MFC가 처음으로 호출하는 함수는 DeleteContents 함수이다. 만약 작업 중에 힙에 할당한 메모리가 있다면 DeleteContents 함수는 이를 해제하는 가장 좋은 장소가 된다. 한 가지 주의할 점은 다큐먼트의 데이터를 소멸자에서 해제해서는 안된다는 점이다. SDI의 경우에는 다큐먼트 개체가 재사용되기 때문에 할당한 메모리가 해제되지 않는다.

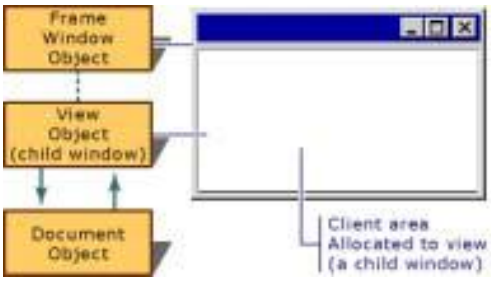
뷰 클래스의 경우에는 소멸자에서 힙에 할당된 메모리를 해제하면 된다. SDI의 경우에도 다큐먼트 클래스는 재사용되지만 뷰는 재사용되지 않는다.

1.3.4 프레임 윈도우

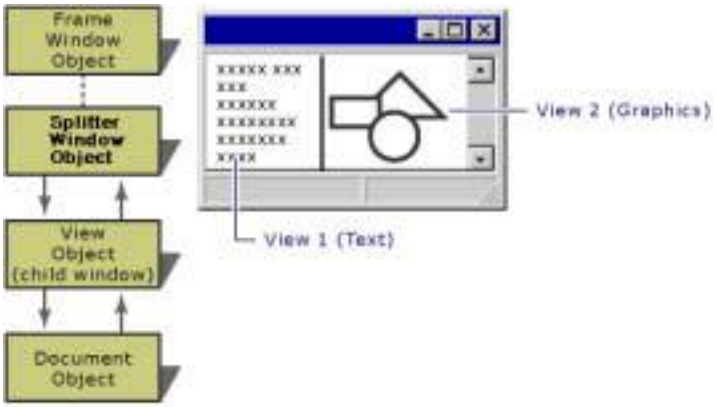
윈도우 기반의 응용 프로그램을 실행할 때, 사용자는 프레임 윈도우에 표시된 다큐먼트와 상호작용을 하게 된다. 다큐먼트 프레임 윈도우는 프레임 자체와 프레임 내의 내용의 2가지 요소로 구성된다.

다큐먼트 프레임 윈도우는 단일 다큐먼트 인터페이스(SDI) 프레임 윈도우와 다중 다큐먼트 인터페이스(MDI) 차일드 윈도우가 있다. MFC는 뷰를 관리하기 위해 프레임 윈도우를 사용한다. 즉, 프레임 자체와 그 내부의 내용은 각기 별개의 MFC 클래스로 구현되고 관리되며, 뷰 윈도우는 프레임 윈도우의 차일드 윈도우이다.

다큐먼트와 관련된 사용자와의 상호작용은 뷰의 클라이언트 영역에서 일어나며, 프레임 윈도우는 뷰에 프레임을 추가하고, 타이틀바, 메뉴, 최소/최대화 버튼 등을 관리한다. 다음 그림은 프레임 윈도우와 뷰의 관계를 나타낸 것이다.



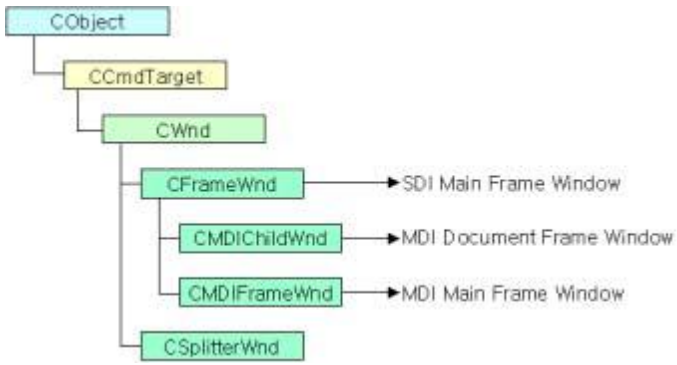
프레임 윈도우에서 여러 뷰를 관리하기 위해 많이 사용하는 방식이 분할 윈도우(splitter window)이다. 분할 윈도우에서 프레임 윈도우의 클라이언트 영역은 분할 윈도우가 차지하며, 분할 윈도우의 분할된 영역들은 각기 페인(pane)이라 불리는 뷰들이 차지한다. 뷰들은 분할 윈도우의 차일드 윈도우이다.



가) 프레임 윈도우 클래스

각 응용 프로그램은 하나의 메인 프레임 윈도우를 가진다. 또한 각 다큐먼트들도 다큐먼트 프레임 윈도우를 가진다. 다큐먼트 프레임 윈도우는 다큐먼트의 데이터들을 나타내기 위해 적어도 하나의 뷰를 가진다.

SDI 형식의 응용 프로그램은 CFrameWnd 클래스에서 유도된 하나의 프레임 윈도우를 가진다. 이 윈도우는 메인 프레임 윈도우인 동시에 다큐먼트 프레임 윈도우 역할을 한다. 이에 비해 MDI 형식의 응용 프로그램에서는 CMDIFrameWnd 클래스에서 유도된 메인 프레임 윈도우와 CMDIChildWnd 클래스에서 유도된 다큐먼트 프레임 윈도우를 가진다.



응용 프로그램 마법사로 프로그램의 골격을 생성하면 마법사는 응용 프로그램을 위한 메인 프레임 윈도우와 MDI 형식의 경우에는 별도로 다큐먼트 프레임 윈도우를 만들어준다. 프레임 윈도우는 뷰를 둘러싸고 있는 프레임을 제공하는 것 이외에도 다양한 일을 한다. 예를 들어 차일드 윈도우를 배치하는 것도 프레임 윈도우의 몫이다. 차일드 윈도우에 포함되는 차일드 윈도우로는 컨트롤 바, 뷰, 그리고 클라이언트 영역 내부에 있는 윈도우나 컨트롤이 포함된다.

또한 프레임 윈도우는 명령을 뷰로 전달하고 컨트롤 윈도우들로부터 통지 메시지를 받을 수 있다.

나) 프레임 윈도우 스타일

프레임 윈도우 클래스의 등록은 WinMain 함수에서 수행된다. 하지만 일반적으로 MFC에서는 WinMain 함수를 수정할 수 없기 때문에 MFC가 지정한 디폴트 윈도우 스타일을 클래스 등록 과정에서 수정할 수는 없다. 비록 마법사가 생성해주는 프레임 윈도우가 대부분의 응용 프로그램에서 수정 없이 사용할 수 있지만, 수정을 원할 경우에는 다음 방법을 통해 가능하다.

Visual C++ 2.0 이후 버전에서는 마법사가 제공하는 프레임 윈도우의 스타일을 프로젝트 생성 시에 수정할 수 있다. 프로젝트를 생성할 때 마법사의 사용자 인터페이스 페이지를 열어보면 메인 프레임과 차일드 프레임의 속성들이 나열되어 있는 것을 볼 수 있다. 이 값들을 수정함으로써 프레임 윈도우의 스타일을 변경할 수 있다. 메인 프레임 윈도우의 경우 분할 윈도우를 사용할 것인지의 여부도 여기에서 지정할 수 있다.

마법사가 이미 골격 생성을 끝낸 경우에는 PreCreateWindow 가상 함수의 오버라이딩을 통해 스타일을 수정할 수 있다. PreCreateWindow 함수는 윈도우가 생성되기 직전에 호출되어 CDocTemplate 클래스가 내부적으로 관리하는 윈도우 생성 작업을 제어할 수 있도록 해준다.

PreCreateWindow 함수의 파라미터인 CREATESTRUCT 구조체 값을 수정함으로써 응용 프로그램은 메인 프레임 윈도우의 속성을 변경할 수 있다.

예를 들어 MDI 메인 프레임 윈도우에 WS_HSCROLL 스타일과 WS_VSCROLL 스타일을 지정하면 클라이언트 영역에서 차일드 윈도우의 위치에 따라 수직 및 수평 스크롤바가 나타난다. 뷰의 내용이 긴 경우 스크롤바가 나타나도록 하기 위해서는 메인 프레임이 아닌 다큐먼트 프레임 윈도우에 위의 스타일을 지정하여야 한다.

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
if( !CMDIFrameWnd::PreCreateWindow(cs) )
return FALSE;
// TODO: CREATESTRUCT cs를 수정하여 여기에서
```

```
// Window 클래스 또는 스타일을 수정합니다.
cs.style = cs.style | WS_HSCROLL | WS_VSCROLL;
return TRUE;
}
```

*** SDI 프레임 윈도우 스타일의 수정**

SDI 응용 프로그램에서 프레임 윈도우의 디폴트 스타일은 WS_OVERLAPPEDWINDOW 스타일과 FWS_ADDTOTITLE 스타일의 조합으로 표시된다. 이 중 FWS_ADDTOTITLE 스타일은 MFC에서만 사용할 수 있는 스타일로 윈도우의 타이틀바에 다큐먼트의 제목을 추가한다. SDI 프레임 윈도우의 스타일을 수정하기 위해서는 CFrameWnd 클래스의 유도 클래스에서 PreCreateWindow 함수를 오버라이드하면 된다.

다음은 크기 조절이 불가능한 경계선을 가지며 최소화 및 최대화 버튼이 없는 윈도우를 생성하는 예이다. 또한 윈도우가 시작될 때 화면 중앙에 나타나도록 한다.

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
// 최대/최소화 버튼과 크기 조절 가능한 경계선이 없는 윈도우 생성
cs.style = WS_OVERLAPPED | WS_SYSMENU | WS_BORDER;
// 화면 크기의 1/3로 윈도우 크기를 설정하고 화면 중앙에 위치시킴
cs.cy = ::GetSystemMetrics(SM_CYSCREEN) / 3;
cs.cx = ::GetSystemMetrics(SM_CXSCREEN) / 3;
cs.y = ((cs.cy * 3) - cs.cy) / 2;
cs.x = ((cs.cx * 3) - cs.cx) / 2;
// 기반 클래스의 함수를 호출
return CFrameWnd::PreCreateWindow(cs);
}
```

*** MDI 차일드 윈도우 스타일의 수정**

MDI 차일드 윈도우의 디폴트 스타일은 WS_CHILD, WS_OVERLAPPEDWINDOW, 그리고 FWS_ADDTOTITLE 스타일의 조합으로 표시된다. 차일드 윈도우의 스타일을 수정하기 위해서는 위의 SDI의 경우와 마찬가지로 PreCreateWindow 함수를 오버라이드하면 된다. 다음은 최대화 버튼이 없는 차일드 윈도우를 생성하는 예이다.

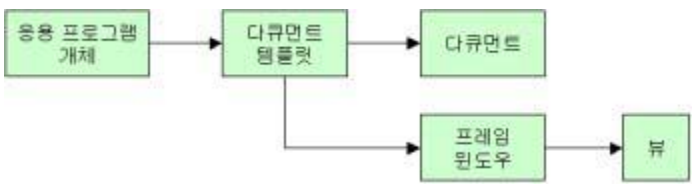
```
BOOL CMyChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
// 최대화 버튼이 없는 차일드 윈도우 생성
cs.style &= ~WS_MAXIMIZEBOX;
// 기반 클래스의 함수를 호출
return CMDIChildWnd::PreCreateWindow(cs);
}
```

다) 프레임 윈도우의 이용

MFC는 뷰와 다큐먼트에 더불어 다큐먼트 프레임 윈도우를 파일 메뉴의 '새 파일'과 '열기' 구현을 위해 이용한다. MFC가 제공하는 프레임 윈도우의 기능은 대부분 수정 없이 사용할 수 있다. 하지만 특별한 목적을 위해서는 프레임 윈도우와 차일드 윈도우를 수정할 수 있다.

*** 다큐먼트 프레임 윈도우 생성**

응용 프로그램을 위해 반드시 필요한 개체로는 응용 프로그램 개체가 있고 다큐먼트/뷰 구조를 이용하는 경우에는 다큐먼트 템플릿, 다큐먼트, 프레임 윈도우, 뷰 등이 있다. 이들이 생성되는 순서는 다음과 같다.



다큐먼트/뷰 구조의 생성은 CDocTemplate 개체가 프레임 윈도우, 다큐먼트, 뷰 개체를 생성하고 연결함으로써 이루어진다. CDocTemplate 개체의 생성자로 전달되는 3개의

CRuntimeClass 클래스 인자는 각기 프레임 윈도우, 다큐먼트, 뷰 클래스를 나타내며, MDI 형식의 응용 프로그램에서 사용자가 '새 파일'이나 '파일 열기' 메뉴를 선택하였을 때 이들 개체들을 동적으로 생성하여 연결시켜 준다.

다큐먼트 템플릿은 이러한 개체들 간의 관계를 저장하였다가 뷰와 다큐먼트를 위해 프레임 윈도우를 생성할 때 이용한다.

RUNTIME_CLASS 메카니즘이 올바르게 동작하기 위해서 프레임 윈도우 클래스에는 DECLARE_DYNCREATE 매크로가 선언되어 있어야 한다. 이 매크로는 CObject 클래스의 동적 생성 메카니즘을 이용하여 다큐먼트 템플릿 윈도우가 생성될 수 있도록 해준다.

사용자가 다큐먼트를 생성하는 명령을 선택하면, MFC는 다큐먼트 템플릿이 다큐먼트 개체, 뷰, 그리고 프레임 윈도우를 생성하도록 한다. 다큐먼트 프레임 윈도우가 생성할 때 SDI의 경우는 CFrameWnd 클래스의 유도 클래스를 MDI의 경우에는 CMDIChildWnd 클래스의 유도 클래스를 생성한다. 이후 MFC는 프레임 윈도우 개체의 LoadFrame 멤버 함수를 사용하여 리소스에서 윈도우 생성에 필요한 정보를 얻어온다.

윈도우가 생성될 때 윈도우의 핸들은 프레임 윈도우 개체와 연결된다. 이후 다큐먼트 프레임 윈도우의 차일드 윈도우로 뷰가 생성된다.

한 가지 주의할 점은 프레임 윈도우가 생성된 이후에 뷰가 생성된다는 점이다. 일반적으로 CWnd 유도 클래스의 생성자에서는 차일드 윈도우를 생성할 수 없다. 생성자가 호출된 시점에서는 아직 CWnd 개체와 연관된 윈도우가 생성되지 않아서 HWND가 유효하지 않기 때문이다. 따라서 차일드 윈도우의 생성과 같은 윈도우와 관련된 초기화는 OnCreate 메시지 핸들러에서 해주는 것이 일반적이다. 이 함수는 윈도우가 생성된 이후 화면에 나타나기 전에 호출된다.

*** 프레임 윈도우의 파괴**

MFC는 다큐먼트/뷰 구조와 관련하여 윈도우의 생성뿐만이 아니라 파괴도 관리한다. 하지만 추가적인 윈도우를 생성한 경우에 그 파괴의 책임은 프로그래머에게 있다.

MFC에서 사용자가 프레임 윈도우를 닫으면 WM_CLOSE 메시지가 발생하고 OnClose 핸들러가 호출된다. 이 핸들러의 기본 구현은 DestroyWindow 함수를 호출하여 윈도우를 파괴한다.

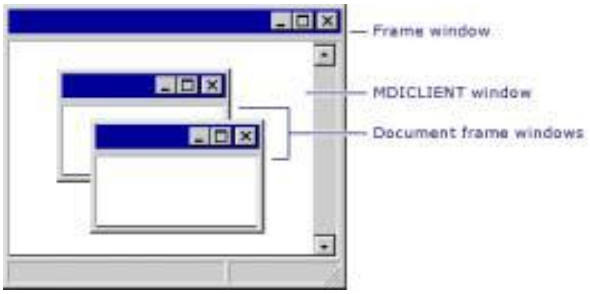
윈도우가 파괴될 때 마지막으로 호출되는 함수는 OnNcDestroy 함수로 윈도우의 비/클라이언트 영역이 파괴된 후 WM_NCDESTROY 메시지의 핸들러로 호출된다. 윈도우가 파괴된 이후에 OnNcDestroy 함수는 PostNcDestroy 함수를 호출한다. 프레임 윈도우에서 PostNcDestroy 함수의 기본 구현은 C++ 윈도우 개체를 삭제한다. 따라서 프레임 윈도우를 파괴할 때는 C++의 delete 연산자를 사용해서는 안되며 반드시 DestroyWindow 함수를 사용하여야 한다.

메인 윈도우 프레임이 종료되면 응용 프로그램도 종료된다. 이 때 수정된 후 저장되지 않은 다큐먼트가 있으면 MFC는 메시지 박스를 통해 사용자에게 저장 여부를 확인하도록 할 수 있다.

*** MDI 차일드 윈도우**

MDI 메인 프레임 윈도우는 MDICLIENT 윈도우라고 불리는 특별한 차일드 윈도우를 포함하고 있다. MDICLIENT 윈도우는 메인 프레임 윈도우의 클라이언트 영역을 관리하며 그 자신이 CMDIChildWnd 클래스의 유도 클래스인 다큐먼트 프레임 윈도우를 포함하고 있다. 다큐먼트 프레임 윈도우도 프레임 윈도우이므로 이들 역시 차일드 윈도우를 가질 수 있고 차일드 윈도우로 명령을 전달할 수 있다.

MDI 프레임 윈도우에서 프레임 윈도우는 MDICLIENT 윈도우를 관리하고 컨트롤바와 함께 그 위치를 관리하고, MDICLIENT 윈도우는 MDI 차일드 프레임 윈도우를 관리한다. 다음 그림은 이들 윈도우들의 관계를 나타낸 것이다.



MDI 프레임 윈도우는 현재 MDI 차일드 윈도우에 대해 작업을 수행한다. MDI 프레임 윈도우는 자신이 명령을 처리하기 이전에 먼저 차일드 윈도우로 보내어 처리할 기회를 준다.

*** 뷰 관리**

프레임 윈도우의 디폴트 구현 중 일부로서, 프레임 윈도우는 현제 활성 윈도우를 관리한다. 분할 윈도우에서와 같이 프레임 윈도우가 하나 이상의 뷰를 포함하고 있다면 현재 뷰는 가장 최근에 사용된 뷰를 말한다. 활성 뷰는 윈도우의 활성 상태나 입력 포커스와는 무관하다.

활성 뷰가 변하는 경우 MFC는 OnActivateView 멤버 함수를 통해 뷰의 활성 상태 변화를 알려준다. OnActivateView 함수의 *bActivate* 파라미터를 검사함으로써 뷰가 활성화되는지 비활성화되는지 알아낼 수 있다. 기본적으로 OnActivateView 함수는 활성화되는 현재 뷰에 포커스를 설정한다. 뷰가 활성화되거나 비활성화되는 경우 특별한 처리가 필요하다면 OnActivateView 함수를 오버라이드하면 된다. 예를 들어 활성화되거나 비활성화되는 뷰에 특별한 표시를 하기 위해서 이 함수를 오버라이드하여 사용할 수 있다.

프레임 윈도우는 명령 전달 경로에 따라 명령을 먼저 현재의 활성 뷰로 보낸다.

*** 메뉴, 컨트롤바, 가속키 관리**

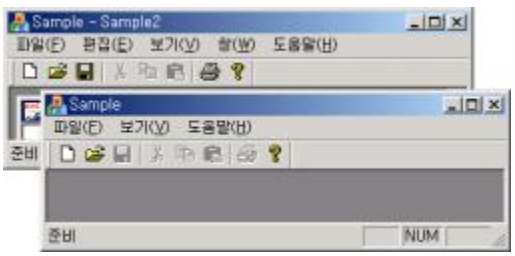
프레임 윈도우는 메뉴, 툴바 버튼, 상태 바, 가속키 등 사용자 인터페이스 개체들의 업데이트를 관리한다.

○ 메뉴 관리

프레임 윈도우는 ON_UPDATE_COMMAND_UI 메카니즘을 사용하여 사용자 인터페이스 개체의 업데이트를 관리한다. 툴바에 있는 버튼과 다른 컨트롤바들은 유휴 처리에서 업데이트된다. 메뉴바에 있는 펼침 메뉴의 메뉴 항목들은 메뉴가 펼쳐지기 직전에 업데이트된다.

MDI 응용 프로그램의 경우 MDI 프레임 윈도우는 메뉴바와 캡션을 관리한다. MDI 프레임 윈도우는 차일드 윈도우가 없을 때 메뉴바로 사용할 디폴트 메뉴를 포함하고 있다. 활성 차일드 윈도우가 있는 경우에 MDI 프레임 윈도우는 활성 MDI 차일드 윈도우를 위한 메뉴로 메뉴바를 교체한다. 만약 MDI 응용 프로그램이 다중 다큐머트 타입을 지원한다면 다큐먼트의 종류에 따라 메뉴와 캡션이 바뀐다. 동일한 다큐먼트 타입의 여러 MDI 차일드 윈도우는 메뉴를 공유한다.

다음 그림은 MDI 응용 프로그램에서 차일드 윈도우가 있는 경우와 없는 경우를 비교한 것이다.



그림을 살펴보면 메뉴와 타이틀바는 물론 툴바의 버튼 상태도 변화한 것을 볼 수 있다. CMDIFrameWnd 클래스는 기본적인 명령에 관한 디폴트 구현을 제공하고 있다. 특히 '새 파일'에 해당하는 ID_WINDOW_NEW 명령은 다큐먼트는 물론 새로운 프레임 윈도우와 뷰를 생성하도록 구현되어 있다.

○ 상태바 관리

프레임 윈도우는 클라이언트 영역 내에서 상태바의 위치와 상태바의 지시자(indicator)들을

관리한다.

프레임 윈도우는 메뉴 항목이나 툴바 버튼을 선택한 경우 상태바의 메시지 영역에 문자열을 출력해준다. 출력되는 문자열은 메뉴 편집기에서 메뉴의 'Prompt' 속성을 통해 지정할 수 있다. 문자열은 두 부분으로 구성되어 있고 '₩n'으로 구분되어 있다. 첫 번째 부분은 상태바에 출력되는 내용이고 두 번째 부분은 툴바 버튼에 대한 풍선 도움말로 출력된다.

기본적으로 마법사는 새로운 메시지를 기다리고 있는 상태에서 상태바에 출력되는 표준 프롬프트인 '준비'에 대한 아이디 AFX_IDS_IDLEMESSAGE를 추가해준다. 또한 마법사에서 문맥 감지 도움말 옵션을 설정하면 '도움말을 보려면 <F1> 키를 누르십시오.'에 대한 아이디 AFX_IDS_IDLEMESSAGE를 추가해준다. 이 내용은 리소스의 문자열 테이블에서 확인할 수 있다.

○ 가속키 관리

프레임 윈도우는 옵션으로 가속키 테이블을 관리하여 가속키 변환을 자동으로 수행해줄 수 있다. 가속키는 메뉴 명령을 발생시킨다. 리소스의 'Accelerator' 항목에 가속키들이 정의되어 있으며, 메뉴의 'Caption' 항목에 가속키에 해당하는 키가 표시되어 있다.

ID	Modifier	Key	Type
ID_CONTEXT_HELP	Shift	VK_F1	VIRTKEY
ID_EDIT_COPY	Ctrl	C	VIRTKEY
ID_EDIT_COPY	Ctrl	VK_INSERT	VIRTKEY
ID_EDIT_CUT	Shift	VK_DELETE	VIRTKEY
ID_EDIT_CUT	Ctrl	X	VIRTKEY
ID_EDIT_PASTE	Ctrl	V	VIRTKEY
ID_EDIT_PASTE	Shift	VK_INSERT	VIRTKEY
ID_EDIT_UNDO	Alt	VK_BACK	VIRTKEY

*** 끌어 놓기 (Drag-and-Drop)**

프레임 윈도우는 윈도우 익스플로러나 파일 관리자와의 관계를 관리한다.
CWinApp 클래스의 InitInstance 함수를 오버라이드하고 메인 프레임 윈도우의 CWnd::DragAcceptFiles 멤버 함수를 호출해줌으로써 프레임 윈도우는 익스플로러나 파일 관리자에서 끌어다 놓은 파일을 열 수 있다.

라) CFrameWnd 클래스

CFrameWnd 클래스는 단일 다큐먼트 인터페이스(SDI)의 오버랩 또는 팝업 프레임 윈도우를 위한 기본 기능을 제공한다. 일반적으로 응용 프로그램에서 프레임 윈도우를 생성하기 위해서는 CFrameWnd 클래스의 유도 클래스를 사용한다.
프레임 윈도우는 두 단계로 생성됩니다. 먼저 생성자를 통해 CFrameWnd 개체가 생성된다.. 하지만 생성자는 실제 윈도우를 생성하지는 않는다. 실제 윈도우는 Create 함수나 LoadFrame 함수를 통해 생성되어 CFrameWnd 개체에 연결된다. CFrameWnd 개체는 *m_hWnd* 멤버 변수를 통해 윈도우를 억세스할 수 있다.

프레임 윈도우를 생성하는 방법은 3가지가 있다.

- Create 함수를 사용하여 직접 생성하는 방법
- LoadFrame 함수를 사용하여 직접 생성하는 방법
- 다큐먼트 템플릿을 이용하여 간접 생성하는 방법

Create 함수나 LoadFrame 함수를 호출하기 전에 C++의 new 연산자를 사용하여 힙에 프레임 윈도우 개체를 생성하여야 한다. 윈도우 개체를 생성하고 Create 함수를 호출하기 전에 아이콘이나 프레임 윈도우의 스타일을 지정하기 위해 AfxRegisterWndClass 함수를 사용하여 윈도우 클래스를 등록할 수 있다.

프레임 생성에 필요한 파라미터를 직접 전달하기 위해서 Create 함수를 사용하면 됩니다. 이에 비해 LoadFrame 함수는 Create 함수에 비해 적은 수의 파라미터를 필요로 하며, 타이틀, 아이콘, 가속키테이블, 메뉴 등은 리소스에서 얻어와 사용한다. LoadFrame 함수를 통해 리소스를 얻어오기 위해서는 리소스들이 동일한 아이디를 가져야 한다. 기본적으로

IDR_MAINFRAME이라는 아이디를 가진다.

CFrameWnd 개체가 뷰와 다큐먼트를 포함하고 있다면 MFC는 자동적으로 뷰와 다큐먼트를 생성해준다. 응용 프로그램 개체가 생성하는 CDocTemplate 개체가 프레임 윈도우, 뷰, 다큐먼트 개체를 생성하고 뷰와 다큐먼트를 연결하는 역할을 한다. CDocTemplate의 생성자에는 다큐먼트, 프레임, 뷰에 대한 파라미터를 CRuntimeClass 형식으로 지정한다. CRuntimeClass 개체는 새로운 프레임을 동적으로 생성하기 위해 MFC에서 사용한다.

```
// SDI의 경우
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME, // 리소스 아이디
RUNTIME_CLASS(CSDIDoc),
RUNTIME_CLASS(CMainFrame), // 주 SDI 프레임 창입니다.
RUNTIME_CLASS(CSDIView));
// MDI의 경우
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
IDR_MDITYPE, // 리소스 아이디
RUNTIME_CLASS(CMDIDoc),
RUNTIME_CLASS(CChildFrame), // 사용자 지정 MDI 자식 프레임입니다.
RUNTIME_CLASS(CMDIView));
```

RUNTIME_CLASS 메카니즘이 올바르게 동작하기 위해서는 CFrameWnd 클래스에서 유도된 프레임 윈도우 클래스에 DECLARE_DYNCREATE 매크로가 선언되어 있어야 한다. CFrameWnd 클래스의 디폴트 구현에는 다음 내용들이 포함되어 있다.

- 프레임 윈도우는 윈도우의 현재 활성 상태나 입력 포커스와 무관하게 현재 활성 뷰를 관리한다. 프레임 윈도우가 활성화되면 활성 뷰는 CView::OnActivateView 함수에 의해 자동으로 활성화된다.

◦ 명령 메시지와 CView 클래스의 OnSetFocus, OnHScroll, OnVScroll 등의 함수에서 처리 되는 많은 프레임 통지 메시지들은 프레임 윈도우가 현재 활성 뷰로 전달해 준다.

◦ 현재의 활성 뷰나 MDI의 경우에는 현재의 활성 차일드 프레임 윈도우가 프레임 윈도우의 캡션을 설정할 수 있다. 이 기능은 프레임 윈도우의 FWS_ADDTOTITLE 스타일의 설정 여부에 따른다.

◦ 프레임 윈도우는 프레임 윈도우의 클라이언트 영역 내에 존재하는 컨트롤 바, 뷰, 그리고 다른 차일드 윈도우들의 위치를 관리한다. 또한 프레임 윈도우는 유휴 시간에 툴바와 컨트롤 바의 버튼을 업데이트하며, 툴바와 상태바를 보이거나 보이지 않게 하는 디폴트 구현을 제공한다.

◦ 프레임 윈도우는 메인 메뉴 바를 관리한다. 또한 팝업 메뉴가 보여지는 경우 프레임 윈도우는 UPDATE_COMMAND_UI 메카니즘을 통해 메뉴의 인에이블, 디스에이블, 체크 등의 상태를 결정한다. 사용자가 메뉴 항목을 선택하면 프레임 윈도우는 상태바에 선택됨 항목에 대한 메시지를 보여준다.

◦ 프레임 윈도우는 가속키 테이블을 옵션으로 가질 수 있다.

◦ 프레임 윈도우는 문맥 감지 도움말에 사용되는 도움말 아이디를 옵션으로 가질 수 있다. 프레임 윈도우는 문맥 감지 도움말(SHIFT+F1)이나 인쇄 미리보기 모드 등 세미모달 상태를 제어할 책임을 진다.

◦ 프레임 윈도우는 파일 관리자에서 끌어다 놓은 파일을 열 수 있다. 또한 응용 프로그램에 파일 확장자가 등록된 경우 프레임 윈도우는 사용자의 더블 클릭이나 ShellExecute 함수에 의해 발생하는 DDE의 열기 명령을 처리한다.

◦ 프레임 윈도우가 응용 프로그램의 메인 윈도우(CWinThread::m_pMainWnd)인 경우, 사용자가 응용 프로그램을 종료하면 프레임 윈도우는 수정된 다큐먼트를 저장할 것인지를 물어본다.

◦ 프레임 윈도우가 응용 프로그램의 메인 윈도우인 경우, 프레임 윈도우는 WinHelp를

실행하는 컨텍스트가 된다. 프레임 윈도우를 닫으면 실행 중인 WinHelp도 종료된다.

마) CMDIFrameWnd 클래스

CMDIFrameWnd 클래스는 다중 다큐먼트 인터페이스(MDI) 프레임 윈도우의 기본 기능을 제공한다. 일반적으로 응용 프로그램에서 MDI 프레임 윈도우를 생성하기 위해서는 CMDIFrameWnd 클래스의 유도 클래스를 사용한다.

MDIFrameWnd 클래스는 CFrameWnd 클래스의 유도 클래스이지만 CMDIFrameWnd 유도 클래스들은 동적으로 생성하지 않기 때문에 DECLARE_DYNCREATE 매크로를 선언하지 않아도 된다.
CMDIFrameWnd 클래스는 기본 구현을 CFrameWnd 클래스로부터 상속하고 있다. 여기에 다음 특성들을 추가적으로 가진다.

- MDI 프레임 윈도우는 MDICLIENT 윈도우를 관리하고 컨트롤바와 함께 그 위치를 지정한다. MDI 클라이언트 윈도우는 MDI 차일드 프레임 윈도우의 부모 윈도우이다.

- MDI 프레임 윈도우는 디폴트 메뉴를 가지고 있어서 활성 차일드 윈도우가 없는 경우 메뉴 바에 표시된다. 활성 MDI 차일드 윈도우가 있는 경우 MDI 프레임 윈도우의 메뉴 바는 MDI 차일드 윈도우의 메뉴로 교체된다.

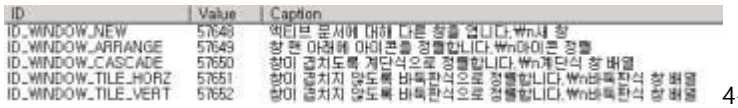
- MDI 프레임 윈도우는 현재 MDI 차일드 윈도우와 상호작용한다. 예를 들어 명령 메시지는 MDI 프레임 윈도우가 처리하기 이전에 현재 활성 MDI 차일드 윈도우로 그 처리를 넘긴다.

- MDI 프레임 윈도우는 차일드 윈도우들을 배열하기 위한 표준 윈도우 명령들을 처리하는 핸들러 함수를 가진다.

    - ID_WINDOW_TILE_VERT
    - ID_WINDOW_TILE_HORZ
    - ID_WINDOW_CASCADE

▫ ID_WINDOW_ARRANGE

▫ MDI 프레임 윈도우는 현재 활성 다큐먼트에 대해 프레임과 뷰를 생성하는 ID_WINDOW_NEW 명령에 대한 기본 구현을 포함하고 있다.



ID	Value	Caption
ID_WINDOW_NEW	57648	액티브 문서에 대해 다른 창을 엽니다.₩n새 창
ID_WINDOW_ARRANGE	57649	창 맨 아래에 아이콘을 정렬합니다.₩n아이콘 정렬
ID_WINDOW_CASCADE	57650	창이 겹치도록 계단식으로 정렬합니다.₩n계단식 창 배열
ID_WINDOW_TILE_HORZ	57651	창이 겹치지 않도록 바둑판식으로 정렬합니다.₩n바둑판식 창 배열
ID_WINDOW_TILE_VERT	57652	창이 겹치지 않도록 바둑판식으로 정렬합니다.₩n바둑판식 창 배열

4-

바) CMDIChildWnd 클래스

CMDIChildWnd 클래스는 다중 다큐먼트 인터페이스 차일드 윈도우의 기본 기능을 제공한다. 일반적으로 응용 프로그램에서 MDI 파일드 윈도우를 생성하기 위해서는 CMDIChildWnd 클래스의 유도 클래스를 사용한다.

MDI 차일드 윈도우는 프레임 윈도우와 비슷하게 보이지만 몇 가지 차이점이 있다. MDI 차일드 윈도우는 데스트탑이 아닌 MDI 프레임 윈도우의 클라이언트 영역 내부에 나타나며, 자신의 메뉴바를 가지지 않고 MDI 프레임 윈도우의 메뉴를 공유한다. 또한 MFC는 현재의 활성 MDI 차일드 윈도우를 위해서 프레임 윈도우의 메뉴를 자동으로 수정한다. CMDIChildWnd 개체가 뷰와 다큐먼트롤 포함하고 있다면 CFrameWnd 개체의 경우에서처럼 MFC는 자동적으로 뷰와 다큐먼트를 생성하고 이들을 연결시켜 준다.

CMDIChildWnd 클래스는 기본 구현을 CFrameWnd 클래스로부터 상속하고 있다. 여기에 다음 특성들을 추가적으로 가진다.

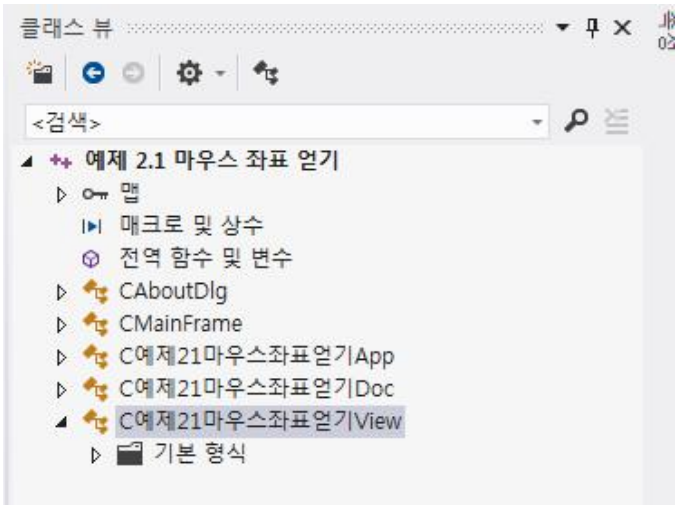
◦ CMultiDocTemplate 클래스와 함께 사용되어 동일한 템플릿에서 생성된 CMDIChildWnd 개체는 메뉴를 공유한다.

◦ 현재 활성 MDI 차일드 윈도우의 메뉴는 MDI 프레임 윈도우의 메뉴를 대신한다. 또한 현재 활성 MDI 차일드 윈도우의 타이틀이 프레임 윈도우의 타이틀에 추가된다.

2. MFC 기반 다지기(1)

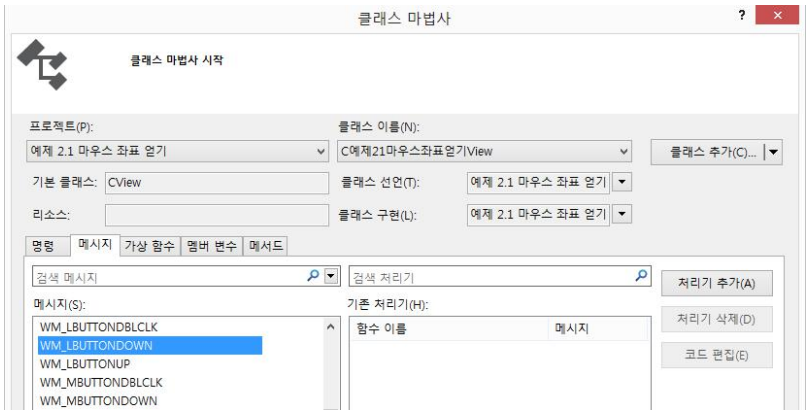
2.1 프로젝트 생성 및 마법사 사용하기

1) MFC 응용 프로그램을 생성하면 아래와 같이 5개의 파생 클래스가 생성된다.



2) 만약 프로그램에 필요한 관련 가상 함수 또는 이벤트 핸들러를 등록하기 원한다면 클

래스 마법사를 사용한다.



3) 클래스로 원하는 메시지를 위의 그림처럼 선택하면 관련 코드가 자동으로 생성된다.

```
// 예제 2.1 마우스 좌표 얻기View.h : C예제21마우스좌표얻기View 클래스의 인터페이스


class C예제21마우스좌표얻기View : public CView
{
public:
	afx_msg void OnLButtonDown(UINT nFlags, CPoint point);  // 1) 함수 선언부
};
```

```
// 예제 2.1 마우스 좌표 얻기View.cpp : C예제21마우스좌표얻기View 클래스의 구현


//2) 함수 정의부
void C예제21마우스좌표얻기View::OnLButtonDown(UINT nFlags, CPoint point)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
	CView::OnLButtonDown(nFlags, point);
}
```

```
// 예제 2.1 마우스 좌표 얻기View.cpp : C예제21마우스좌표얻기View 클래스의 구현


BEGIN_MESSAGE_MAP(C예제21마우스좌표얻기View, CView)
	ON_WM_LBUTTONDOWN()
```

```
END_MESSAGE_MAP()
```

2.2 마우스 메시지 처리

마우스 관련 메시지는 크게 두 가지로 구분할 수 있다.

클라이언트 영역(사용자 영역)에 대한 마우스 메시지

넌클라이언트 영역(비 사용자 영역)에 대한 마우스 메시지

2.2.1 클라이언트 영역(사용자 영역)에 대한 마우스 메시지

클라이언트 영역의 마우스 메시지는 뷰 윈도우 영역 내에 커서가 있을 때 윈도우즈 OS에 의해 발생되며, 그 종류는 다음 표와 같다.

메시지 유형	의미
WM_MOUSEMOVE	마우스를 움직일 때
WM_LBUTTONDBCLK	마우스 좌측버튼을 더블 클릭할 때
WM_LBUTTONDOWN	마우스 좌측버튼을 클릭했을때
WM_LBUTTONUP	마우스 좌측버튼을 놓을 때
WM_MBUTTONDBCLK	마우스 중간버튼을 더블 클릭할 때
WM_MBUTTONDOWN	마우스 중간버튼을 클릭했을때
WM_MBUTTONUP	마우스 중간버튼을 놓을 때
WM_RBUTTONDBCLK	마우스 우측버튼을 더블 클릭할 때
WM_RBUTTONDOWN	마우스 우측버튼을 클릭했을때
WM_RBUTTONUP	마우스 우측버튼을 놓을 때

메시지 핸들러 원형

afx_msg void OnLButtonDown(UINT nFlags, CPoint point);

nFlags : Shift 키나 Ctrl 키 등이 눌렸는지 여부

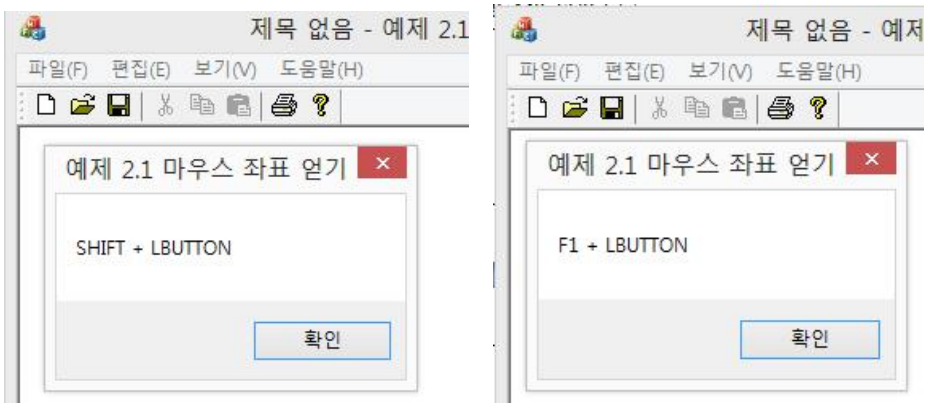
MK_SHIFT, MK_CONTRO, MK_LBUTTON 등으로 확인할 수 있다.

point : 마우스 커서의 좌표(클라이언트 좌표)

마우스 메시지를 처리하기 위해서는 클래스 마법사를 통해 원하는 마우스 메시지를 등록한다.

```
void C예제21마우스좌표얻기View::OnLButtonDown(UINT nFlags, CPoint point)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
	// 1st 파라미터 조사하기 , shift, control과 3개마우스 버튼만 가능
	if ( nFlags & MK_SHIFT )
		MessageBox(TEXT("SHIFT + LBUTTON"));

	// 그외의 키보드 조사하기.
	if ( GetKeyState( VK_F1 ) < 0 ) // 눌린경우 음수 리턴.
		MessageBox(TEXT("F1 + LBUTTON"));

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

예제 2.1 마우스 좌표 얻기



위 결과 화면 중 왼쪽 이미지는 사용자가 SHIFT와 마우스 왼쪽 버튼을 클릭한 결과이고 오른쪽 이미지는 사용자가 F1 과 마우스 왼쪽 버튼을 클릭한 결과이다.

아래 예제는 마우스 캡쳐 관련 내용이다.

```
class C예제22마우스캡쳐View : public CView
{
private:
        CPoint m_ptFrom;
        CPoint m_ptTo;
        bool   m_bTracking;
```

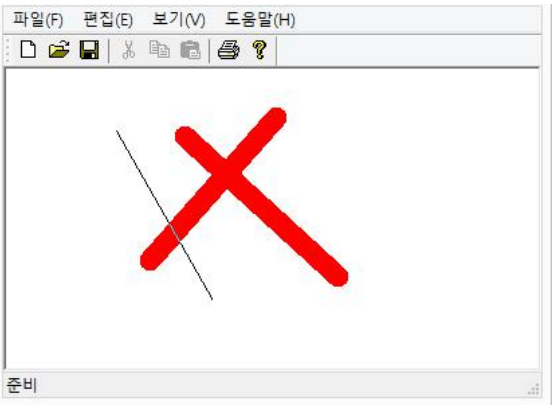
```
C예제22마우스캡쳐View::C예제22마우스캡쳐View()
{
        // TODO: 여기에 생성 코드를 추가합니다.
        m_bTracking = FALSE;
}
```

```
void C예제22마우스캡쳐View::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        //앵커 위치를 기록하고 추적 플래그 설정
        m_ptFrom = point;
        m_ptTo           = point;
        m_bTracking      = TRUE;
        // 캡쳐가 가능하면 마우스 캡쳐
        if( GetParentFrame()->m_bAutoMenuEnable )
                SetCapture();
        CView::OnLButtonDown(nFlags, point);
}


void C예제22마우스캡쳐View::OnLButtonUp(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        if( m_bTracking)  {
```

```
                m_bTracking       = FALSE;
                if( GetCapture() == this)
                        ::ReleaseCapture();
                CClientDC         dc(this);
                InvertLine(&dc, m_ptFrom, m_ptTo);
                CPen pen(PS_SOLID, 16, RGB(255, 0, 0));
                dc.SelectObject(&pen);
                dc.MoveTo(m_ptFrom);
                dc.LineTo(point);
        }
        CView::OnLButtonUp(nFlags, point);
}

void C예제22마우스캡쳐View::OnMouseMove(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        if( m_bTracking)  {
                CClientDC dc(this);
                InvertLine(&dc, m_ptFrom, m_ptTo);
                InvertLine(&dc, m_ptFrom, point);
                m_ptTo   = point;
        }
        CView::OnMouseMove(nFlags, point);
}

void C예제22마우스캡쳐View::InvertLine(CDC * pDC, CPoint ptFrom, CPoint ptTo)
{
        int nOldMode = pDC->SetROP2(R2_NOT);
        pDC->MoveTo(ptFrom);
        pDC->LineTo(ptTo);
        pDC->SetROP2( nOldMode);
```

```
}
```

예제 2.2 마우스 캡쳐



2.2.2 넌클라이언트 영역(사용자 영역)에 대한 마우스 메시지

넌클라이언트 영역의 마우스 메시지는 사용자 영역을 제외한 메뉴바, 툴바, 상태바, 캡션바 등 메인 프레임 클래스가 관리하는 영역에 커서가 있을 때 윈도우즈 OS에 의해 발생되며 그 종류는 다음 표와 같다.

point : 마우스 커서의 좌표(클라이언트 좌표)

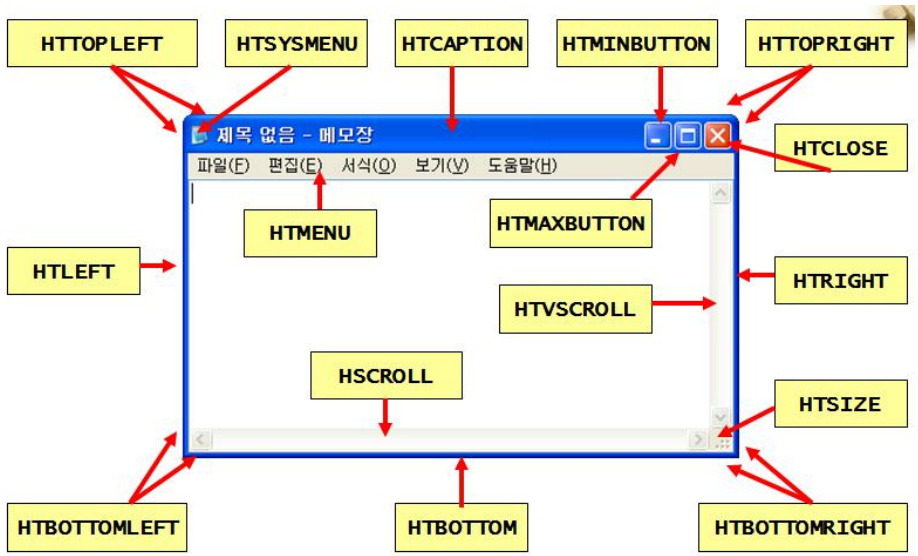
메시지 유형	의미
WM_NCLBUTTONDBCLK	넌클라이언트 영역에서 마우스 좌측버튼을 더블클릭했을 때
WM_NCLBUTTONDOWN	넌클라이언트 영역에서 마우스 좌측버튼을 클릭했을때
WM_NCLBUTTONUP	넌클라이언트 영역에서 마우스 좌측버튼을 놓을 때
WM_NCMOUSEMOVE	넌클라이언트 영역에서 사용자가 마우스를 움직일 때 발생

메시지 핸들러 원형

afx_msg void OnNCLButtonDown(UINT nHitTest, CPoint point);

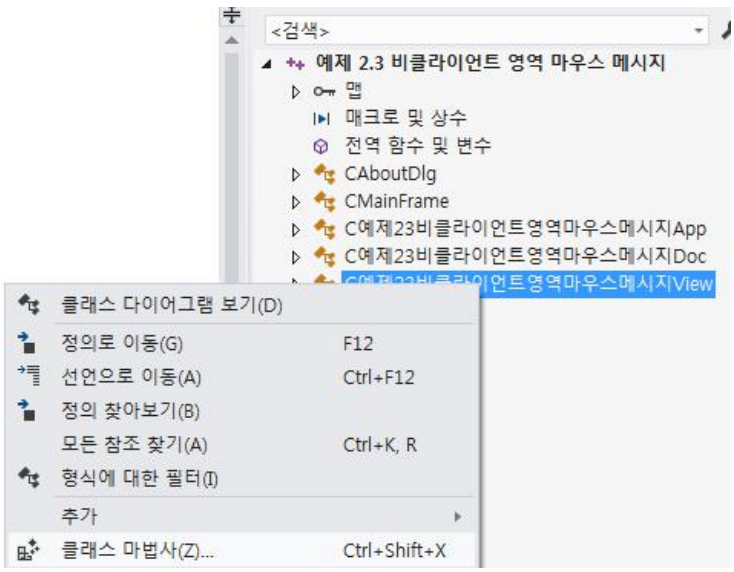
nHitTest: 커서가 넌클라이언트 영역의 어느 위치에 있는지를 나타내는
히트테스트 코드

point : 마우스 커서의 좌표(스크린 좌표 좌표)

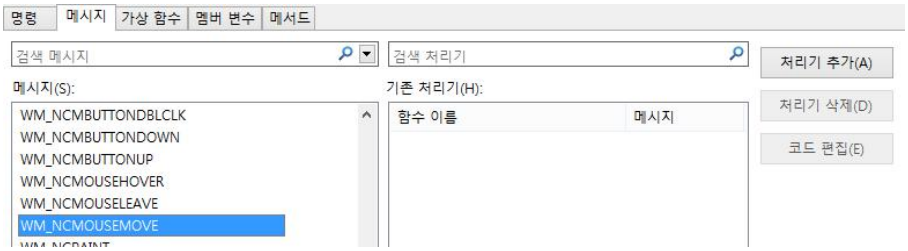


*** 비클라이언트 영역 마우스 이벤트 확인예제**

```
// 클래스 마법사를 실행
```



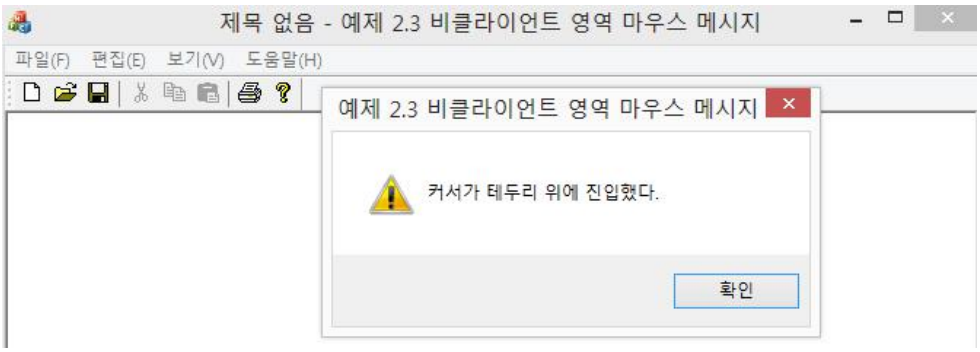
// 클래스 마법사에서 WM_NCMOUSEMOVE 처리기 추가




```
void C예제23비클라이언트영역마우스메시지View::OnNcMouseMove(UINT nHitTest,
CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        CString strInfo ;
        if ( nHitTest == HTBORDER )
        {
                strInfo = "커서가 테두리 위에 진입했다." ;
                AfxMessageBox( strInfo ) ;
        }
```

```
        CView::OnNcMouseMove(nHitTest, point);
}
```

예제 2.3 비클라이언트 영역 마우스 메시지



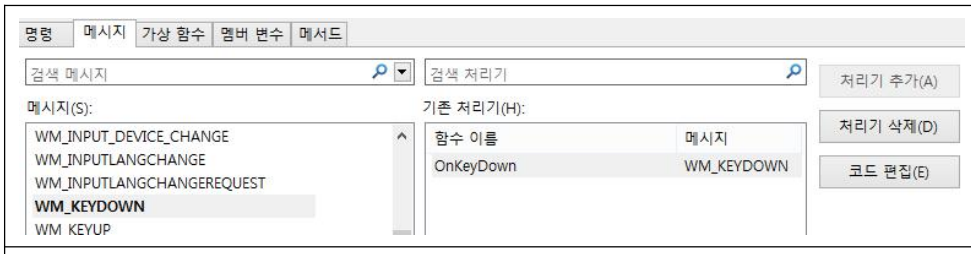
2.2 키보드 메시지 처리

키보드 이벤트에 대해 발생하는 메시지는 WM_KEYDOWN과 WM_KEYUP이다. 결국 하나의 키를 눌렀다 떼면 이 두 메시지가 순서대로 발생한다.

Alt+ 키의 조합은 일반 키 입력으로 취급되지 않고 시스템 키 입력으로 취급된다. 따라서 발생하는 메시지가 달라지는데, 메시지 이름에 SYS가 추가되어서 WM_SYSKEYDOWN, WM_SYSKEYUP이 된다.

*** 키보드로 마우스 처리하기**

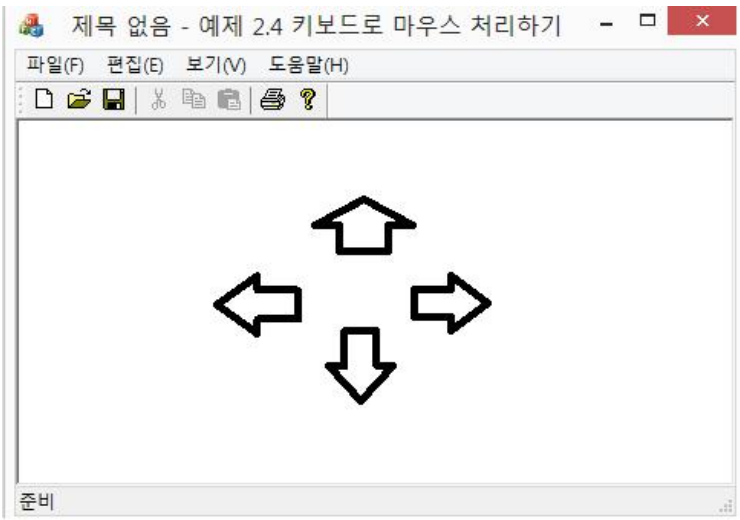
```
// 클래스 마법사 에서 WM_KEYDOWN 처리기 추가
```




```
void C예제24키보드로마우스처리하기View::OnKeyDown(UINT nChar, UINT nRepCnt,
UINT nFlags)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
	CPoint ptCurPos ;
	if ( ::GetCursorPos( &ptCurPos ) )
	{
		switch ( nChar )
		{
		case VK_LEFT  : ptCurPos.x -= 30 ;
					::SetCursorPos( ptCurPos.x ,
ptCurPos.y ) ;
					break ;
		case VK_RIGHT : ptCurPos.x += 30 ;
					::SetCursorPos( ptCurPos.x ,
ptCurPos.y ) ;
					break ;
		case VK_UP    : ptCurPos.y -= 30 ;
					::SetCursorPos( ptCurPos.x ,
ptCurPos.y ) ;
					break ;
		case VK_DOWN  : ptCurPos.y += 30 ;
					::SetCursorPos( ptCurPos.x ,
ptCurPos.y ) ;
					break ;
```

```
                }
        }
        CView::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

예제 2.4 키보드로 마우스 처리하기



*** 캐럿 처리**

캐럿이란 문자를 입력받는 지점을 나타내는데 사용하는 깜박이는 수직 바를 말한다. 프로그램은 하나의 캐럿을 소유할 수 있기 때문에 원하는 대로 생성하여 사용할 수는 없다.

```
class C예제23캐럿View : public CView
{
private:
        CPoint pCaret; // caret의 위치...
        int cxChar; // 글꼴의 너비.
        int cyChar; //        높이.
```

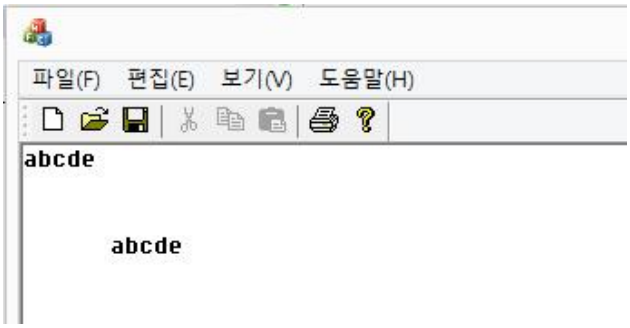
```
C예제23캐럿View::C예제23캐럿View()
{
```

```
	// TODO: 여기에 생성 코드를 추가합니다.
	pCaret.x = 0;
	pCaret.y = 0;
}

void C예제23캐럿View::OnSetFocus(CWnd* pOldWnd)
{
	CView::OnSetFocus(pOldWnd);
	// TODO: 여기에 메시지 처리기 코드를 추가합니다.
	// 캐럿을 생성 한다.
	CreateSolidCaret(10, 20);
	SetCaretPos( pCaret );
	ShowCaret();
}

void C예제23캐럿View::OnKillFocus(CWnd* pNewWnd)
{
	CView::OnKillFocus(pNewWnd);
	// TODO: 여기에 메시지 처리기 코드를 추가합니다.
	// Caret 제거.
	HideCaret();
	DestroyCaret();
}

int C예제23캐럿View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CView::OnCreate(lpCreateStruct) == -1)
		return -1;
	// system FONT( 고정길이 Font의 너비와 높이를 얻는다.)
	CClientDC dc(this);
	// 현재 DC에 OS의 고정길이 폰트를 넣는다.
```

```
        CFont* old = (CFont*)dc.SelectStockObject( SYSTEM_FIXED_FONT );
        TEXTMETRIC tm;
        dc.GetTextMetrics( &tm );
        cxChar = tm.tmAveCharWidth;
        cyChar = tm.tmHeight;
        dc.SelectObject( old );
        return 0;
}

void C예제23캐럿View::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        switch( nChar ) // virtual key code
        {
        case VK_UP:       pCaret.y = max( pCaret.y - cyChar, 0);
        break;
        case VK_DOWN:     pCaret.y = pCaret.y + cyChar;              break;
        case VK_LEFT:     pCaret.x = max( pCaret.x - cxChar, 0);
        break;
        case VK_RIGHT:    pCaret.x = pCaret.x + cxChar;              break;
        }
        SetCaretPos( pCaret );
        CView::OnKeyDown(nChar, nRepCnt, nFlags);
}

void C예제23캐럿View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        if ( nChar == VK_RETURN ) //return 키
        {
                pCaret.y += cyChar;
```

```
                pCaret.x = 0;
                SetCaretPos( pCaret );
                return ;
        }
        CClientDC dc(this);
        CFont* old = (CFont*)dc.SelectStockObject( SYSTEM_FIXED_FONT );

        // 출력전에 caret 감추기.
        HideCaret();
        TCHAR msg[] = { nChar }; // 1개의 문자를 문자열로 바꾸기.
        dc.TextOut( pCaret.x, pCaret.y, msg, 1);
        pCaret.x += cxChar;
        SetCaretPos( pCaret );
        ShowCaret(); // 출력후 다시 Caret를 보여 준다.
        dc.SelectObject( old); // dc 복구.

        CView::OnChar(nChar, nRepCnt, nFlags);
}

void C예제23캐럿View::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        // Caret의 위치를 이동한다....
        pCaret.x = ( point.x / cxChar ) * cxChar;
        pCaret.y = ( point.y / cyChar ) * cyChar;
        SetCaretPos( pCaret );
        CView::OnLButtonDown(nFlags, point);
}
```

예제 2.5 캐럿



*** 문자 입력1**

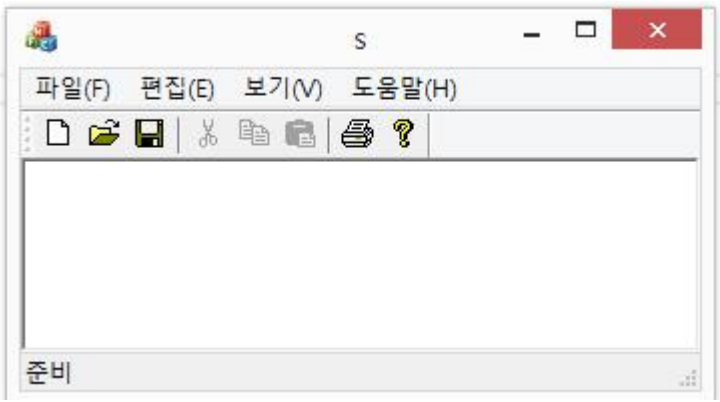
키보드 입력과 관련한 메시지 중에서 우리가 꼭 알아야 할 것이 있는데 그것이 바로 WM_CHAR 메시지이다. 이 메시지는 키보드가 눌렸을 때 그 값이 문자 값이면 발생하는 메시지이다.

WM_CHAR 처리기 추가


```
void C예제26문자입력View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.

	CString temp = TEXT("");

	temp.Format(TEXT("%c"), nChar);
	AfxGetMainWnd()->SetWindowText(temp);
```

```
        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

예제 2.6 문자 입력1

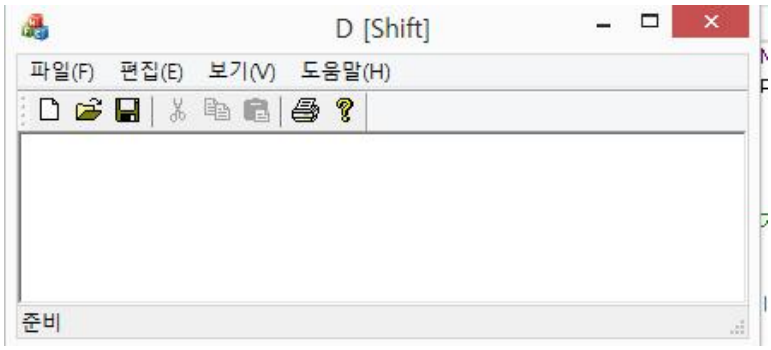


*** 문자 입력2**

```
void C예제27문자입력2View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        CString str = TEXT("");
        str.Format(TEXT("%c"), nChar);

        if(::GetKeyState(VK_SHIFT) < 0 )            str += " [Shift]";
        if(::GetKeyState(VK_CONTROL) < 0 )          str += " [CTRL]";
        if(::GetKeyState(VK_NUMLOCK) < 0 )          str += " [NumLock]";

        AfxGetMainWnd()->SetWindowText(str);

        CView::OnChar(nChar, nRepCnt, nFlags);
```

```
}
```

예제 2.7 문자 입력2



3. MFC 프로그래밍 기반 다지기(II)

3.1 화면 출력하기

3.1.1 디바이스 컨텍스트(Device Context)

우리가 화면에 무언가를 출력하기 위해서는 반드시 윈도우즈 OS로부터 화면을 사용할 수 있는 권한을 얻어야 한다. 문자 하나를 화면에 표시하려고 해도 디바이스 컨텍스트를 얻어야 출력이 가능하다.

CDC : DC에 대한 기초 클래스로 화면이나 프린터 출력에 관계된 대부분의 맴버함수를 포함한다. 출력할 수 있는 영역에 따라 아래의 파생 클래스가 있다.

CWindowDC : 캡션바, 메뉴바, 상태바 등 넌클라이언트 영역을 포함한 전체 윈도우를 표시하는 DC를 관리한다.

CClinetDC : 클라이언트 영역만을 표시하는 DC를 관리한다.

CPaintDC : WM_PAINT 메시지가 발생했을 때 다시 그려져야 할 영역에 대한 DC를 관리하며 WM_PAINT 의 메시지 핸들러인 OnPaint() 함수에서 사용한다.

CMetaFileDC() : 윈도우즈 메타 파일(.WMF나 .EMF)에 대한 DC를 관리한다. 메타 파일이란 그래픽 이미지를 생성해낼 수 있는 GDI 명령들로 구성된 파일을 뜻한다.
메타 파일은 외관상 일반 이미지처럼 보이지만 GDI 명령들의 집합으로 구성되기 때문에 일반 이미지 파일(.BMP, JPG)에 비해 크기가 작고 축소/확대시켜도 계단 현상이 발생되지 않는다.

3.1.2 GDI(Graphics Device Interface)

윈도우즈가 제공하는 GDI는 서로 다른 구조를 지닌 출력 장치에에 대한 정보를 스스로 판단하고 분석하여 실제로 사용해야 할 드라이버를 로드한다.
만약, 화면에 사각형을 출력하려면 Rectangle()이라는 GDI함수를 사용하고 사각형 내부를 색칠하려면 브러시라는 GDI 오브젝트를 사용해야 한다. 이러한 GDI 오브젝트나 함수를 사용하려면 우선 DC를 얻어야 한다.

GDI 오브젝트	클래스	기능	디폴트 속성

			값
펜 오브젝트	CPen	선을 그릴 때 사용된다.	굵기1픽셀 검은 실선
브러쉬 오브젝트	CBrush	특정 영역의 내부를 칠한다.	배경 브러쉬 (보통 흰색)
폰트 오브젝트	CFont	글꼴을 지정해준다.	시스템 폰트
비트맵 오브젝트	CBitmap	비트맵 이미지를 출력한다.	없음
팔레트 오브젝트	CPalette	기존 팔래트를 조절하거나 새로 생성한다	없음
영역 오브젝트	CRgn	다각형이나 원형이 가지는 범위를 설정한다.	없음

사용빈도가 많은 GDI 오브젝트에 대해 윈도우즈는 내장 GDI 오브젝트라는 이름으로 제공한다. 내장 GDI 오브젝트는 시스템에 내장되어 있으므로 일반 GDI 오브젝트와 같이 객체를 생성할 필요가 없다.

펜	브러쉬	폰트
BLACK_PEN (검정 펜)	BLACK_BRUSH (검정 브러쉬)	ANSI_FIXED_FONT (ANSI 고정폰트)
WHITE_PEN 흰색 펜)	WHITE_BRUSH (흰색 브러쉬)	ANSI_VAR_FONT (ANSI 가변 폰트)
NULL_PEN (널 펜)	GRAY_BRUSH (회색 브러쉬)	SYSTEM_FONT (시스템 폰트)

```
// 1. 기본 DC 확인
void C예제31GDI사용하기View::OnLButtonDown(UINT nFlags, CPoint point)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
	CDC* pDC = GetDC();
	pDC->MoveTo(10, 10);
	pDC->LineTo(110, 10);
	ReleaseDC(pDC);
	CView::OnLButtonDown(nFlags, point);
}
```

```
void C예제31GDI사용하기View::OnDraw(CDC* pDC)
```

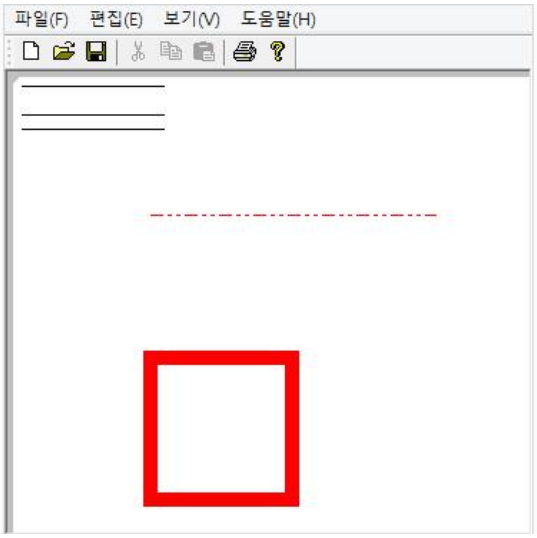
```
{
        C예제31GDI사용하기Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
                return;

   // TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
    // 화면에서 사라지지 않음
    //  WM_PAINT 메시지가 발생할 때마다 자동으로 호출되는 메시지

        pDC->MoveTo(10, 30);        pDC->LineTo(110, 30);
        pDC->MoveTo(10, 40);        pDC->LineTo(110, 40);
        CRect rect;
        GetClientRect(&rect);
        // 네모상자를 그리는 함수(선이 아닌 지정한 색상으로 채워준다.)
        // 시작되는 좌표(x, y), 폭과 높이(10, 10), 지정한 색상)
        pDC->FillSolidRect(rect.left, rect.top, 10, 10, RGB(192, 192, 192));
        CPen pen;
        // 새로운 펜 생성 (흰색)
        pen.CreatePen(PS_SOLID, 1, RGB(255, 255, 255));
        pDC->SelectObject(&pen);
        // 부채꼴 모양으로 그려줌
        pDC->Pie(CRect(0, 0, 40, 40),CPoint(20, 0), CPoint(0, 20));
        pDC->FillSolidRect(0, 0, 4, rect.bottom, RGB(192, 192, 192));
        pDC->FillSolidRect(0, 0, rect.right, 4, RGB(192, 192, 192));
            // 2. Cen
        DrawLine((CPaintDC*)pDC);
}
```

```
// 전형적인 코딩 스타일
void C예제31GDI사용하기View::DrawLine(CPaintDC *pDC)
{
```

```
        //1. 새로운 펜을 생성한다.// PS_DOT 확인
        CPen* pNewPen = new CPen(PS_DASHDOTDOT, 1, RGB(255, 0, 0));
        //2. 본래 DC에 등록된 펜이 무엇인지 저장할 포인터
        CPen* pOldPen = NULL;
        //3. 새로만든 팬을 선택하되 본래의 펜이 무엇인지 기억한다.
        pOldPen = pDC->SelectObject(pNewPen);
        //4. 배경이 투명하도록 지정한 좌표에 선을 긋는다.
        pDC->SetBkMode(TRANSPARENT);
        pDC->MoveTo(100, 100);
        pDC->LineTo(300, 100);
        //5. 본래에 사용하던 펜으로 교체한다.
        pDC->SelectObject(pOldPen);
        //6. 다쓴 펜 객체를 삭제한다.
        delete pNewPen;
}
```

```
void C예제31GDI사용하기View::OnRButtonDown(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        CDC* pDC = GetDC();
        CPen* pNewPen = new CPen(PS_SOLID, 10, RGB(255, 0, 0));
        CPen* pOldPen = NULL;
        pOldPen = pDC->SelectObject(pNewPen);
        pDC->SetBkMode(TRANSPARENT);
        pDC->Rectangle(CRect(100, 200, 200, 300));
        pDC->SelectObject(pOldPen);
        delete pNewPen;
        ReleaseDC(pDC);
        CView::OnRButtonDown(nFlags, point);
}
```

예제 3.1 GDI 사용하기1



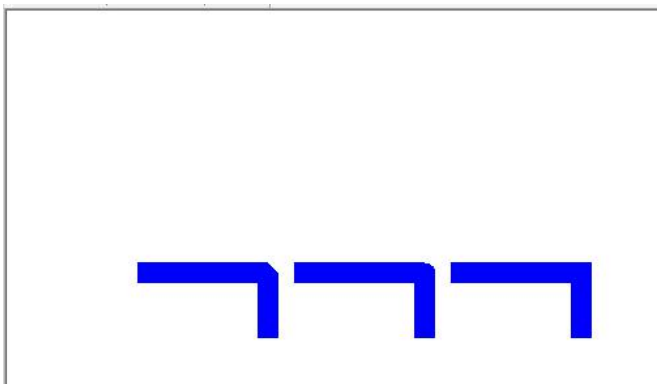

```
void C예제31GDI사용하기View::OnLButtonDown(UINT nFlags, CPoint point)
{
	CDC* pDC = GetDC();
	// 폭과 높이가 정확하게 100
	// 펜의 두께가 바뀌어도 100 을 넘지 않는다.
	CPen* pNewPen = new CPen(PS_SOLID | PS_INSIDEFRAME,		10,
RGB(255, 0, 0));
	CPen* pOldPen = NULL;
	pOldPen = pDC->SelectObject(pNewPen);
	pDC->SetBkMode(TRANSPARENT);
	pDC->Rectangle(CRect(210, 200, 310, 300));
	pDC->SelectObject(pOldPen);
	delete pNewPen;
	ReleaseDC(pDC);
	CView::OnLButtonDown(nFlags, point);
}
```

```
void C예제31GDI사용하기View::OnDraw(CDC* pDC)
{
	C예제31GDI사용하기Doc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;
	// TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
    // 화면에서 사라지지 않음
    //  WM_PAINT 메시지가 발생할 때마다 자동으로 호출되는 메시지
	DrawLine((CPaintDC*)pDC);

}

void C예제31GDI사용하기View::DrawLine(CPaintDC *pDC)
{
	int nStyle[3];
	nStyle[0] = PS_GEOMETRIC | PS_SOLID | PS_ENDCAP_FLAT | PS_JOIN_BEVEL;
	nStyle[1] = PS_GEOMETRIC | PS_SOLID | PS_ENDCAP_FLAT | PS_JOIN_ROUND;
	nStyle[2] = PS_GEOMETRIC | PS_SOLID | PS_ENDCAP_FLAT | PS_JOIN_MITER;
	CPen* pNewPen = NULL;
	CPen* pOldPen = NULL;
	LOGBRUSH lb;
	lb.lbStyle = BS_SOLID;
	lb.lbColor = RGB(0, 0, 255);
	for(int i = 0; i < 3; i++) {
		pNewPen = new CPen(nStyle[i], 16, &lb);
		pOldPen = pDC->SelectObject(pNewPen);
		//ㄱ자 모양으로 선을 그리되 이 두 선을 서로 패스로 묶어준다.
		// BeginPath EndPath의 역할 : 사이의 선이 연결선음을 암시
		pDC->BeginPath();
		pDC->MoveTo(100 + i * 120, 200);
```

```
                pDC->LineTo(200 + i * 120, 200);
                pDC->LineTo(200 + i * 120, 200 + 50);
                pDC->EndPath();
                //두 연결된 선의 꺽어진 부분에 대해 Join 스타일을 적용하여 랜
더링한다.
                // 여기서 그려주게 됨
                pDC->StrokePath();
                pDC->SelectObject(pOldPen);
                delete pNewPen;
        }
}

void C예제31GDI사용하기View::OnRButtonDown(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        CDC* pDC = GetDC();
        int nStyle[3];
        nStyle[0] = PS_GEOMETRIC | PS_SOLID | PS_ENDCAP_ROUND;
        nStyle[1] = PS_GEOMETRIC | PS_SOLID | PS_ENDCAP_SQUARE;
        // 선이 정확하게 넣어준 값 범위에서 그려진다.
        nStyle[2] = PS_GEOMETRIC | PS_SOLID | PS_ENDCAP_FLAT;
        CPen* pNewPen = NULL;
        CPen* pOldPen = NULL;
        //생성할 펜에 대한 정보를 구조체에 넣는다.
        // 펜의 색상과 스타일 정의 가능
        LOGBRUSH lb;
        lb.lbStyle = BS_SOLID;
        lb.lbColor = RGB(0, 0, 255);        //파란색
        for(int i = 0; i < 3; i++) {
                //LOGBRUSH 구조체를 이용하여 펜을 생성한다.
                pNewPen = new CPen(nStyle[i], 16, &lb);
```

```
                pOldPen = pDC->SelectObject(pNewPen);


                pDC->MoveTo(100, 200 + i * 20);
                pDC->LineTo(200, 200 + i * 20);
                pDC->SelectObject(pOldPen);
                delete pNewPen;
        }
        ReleaseDC(pDC);

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

예제 3.2 GDI 사용하기2



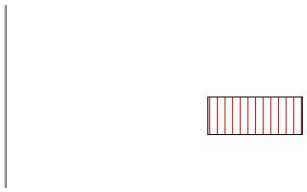
```
void C예제31GDI사용하기View::OnLButtonDown(UINT nFlags, CPoint point)
{
        CDC* pDC = GetDC();
        //녹색의 Solid 브러시를 생성한다.
        CBrush* pNewBrush = new CBrush(RGB(0, 255, 0));
        CBrush* pOldBrush = NULL;
        pOldBrush = pDC->SelectObject(pNewBrush);
        //녹색으로 칠해진 네모가 그려진다.
        pDC->Rectangle(CRect(210, 200, 310, 300));
```

```
	pDC->SelectObject(pOldBrush);
	delete pNewBrush;
	ReleaseDC(pDC);
	CView::OnLButtonDown(nFlags, point);
}

void C예제31GDI사용하기View::OnDraw(CDC* pDC)
{
	C예제31GDI사용하기Doc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;
// TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
    // 화면에서 사라지지 않음
    //  WM_PAINT 메시지가 발생할 때마다 자동으로 호출되는 메시지
	DrawLine((CPaintDC*)pDC);
}

void C예제31GDI사용하기View::DrawLine(CPaintDC *pDC)
{
	//격자 무늬의 해치 브러시를 생성한다.
	CBrush* pNewBrush = new CBrush(HS_VERTICAL, RGB(255, 0, 0));
	CBrush* pOldBrush = NULL;
	pOldBrush = pDC->SelectObject(pNewBrush);
	pDC->SetBkMode(TRANSPARENT);
	//격자 무늬가 입혀진 네모가 그려진다.
	pDC->Rectangle(CRect(210, 200, 310, 240));
	pDC->SelectObject(pOldBrush);
	delete pNewBrush;
	ReleaseDC(pDC);
}
```

```
void C예제31GDI사용하기View::OnRButtonDown(UINT nFlags, CPoint point)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
	CDC* pDC = GetDC();
	CBitmap  Bmp;
	CBrush* pNewBrush = new CBrush;
	CBrush* pOldBrush = NULL;
	//비트맵 파일을 로딩한다.
	Bmp.LoadBitmap(IDB_BITMAP1);
	//비트맵을 가지고 패턴 브러시를 생성한다.
	pNewBrush->CreatePatternBrush(&Bmp);
	pOldBrush = pDC->SelectObject(pNewBrush);
	//비트맵 패턴이 입혀진 네모가 그려진다.
	pDC->Rectangle(CRect(210, 200, 310, 300));
	pDC->SelectObject(pOldBrush);
	delete pNewBrush;
	ReleaseDC(pDC);
	CView::OnRButtonDown(nFlags, point);
}
```

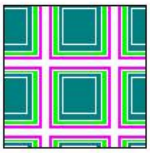
예제 3.3 GDI 사용하기3

프로그램 최초 실행 시 화면



마우스 왼쪽 버튼 클릭시 화면

마우스 오른쪽 버튼 클릭시 화면



4. 사용자 인터페이스(1)

4.1 커서 모양 변경하기

윈도우즈에서 말하는 커서는 마우스의 위치를 알려주는 32*32 픽셀 크기의 비트맵 이미지를 뜻한다. 사용자 정의 커서를 사용하기 위해서는 아래의 단계를 거친다.

1) 리소스 항목에 커서를 입포트한다.
2) 커서를 선택하고 커서의 ID를 설정한다.
3) 새로운 커서가 준비되었으면 핫스팟을 설정한다.
   핫스팟은 커서 이미지상에 위치하는 커서 참조점이다.
4) 커서 모양을 바꾸는 가장 일반적인 방법은 WM_SETCURSOR 메시지에 대해 처리하는 것이다.

윈도우즈 OS는 커서가 윈도우 위를 지날 때 해당 윈도우에 WM_SETCURSOR 메시지를 전달한다. 우리는 이렇게 전달된 WM_SETCURSOR 메시지를 맵핑하여 적절히 처리하면 된다.

```
// C예제41커서View 메시지 처리기

#include "resource.h"
BOOL C예제41커서View::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        ::SetCursor(AfxGetApp()->LoadCursor(IDC_CURSOR1));
        return CView::OnSetCursor(pWnd, nHitTest, message);
}
```

예제 4.1 커서

새로 만든 커서(IDC_CURSOR1)를 로드하기 위해 LoadCursor()를 사용하고 있는데 이상한 것은 단독으로 호출되지 않고 AfxGeApp()를 통해 호출되고 있다.

예제에서 사용된 LoadCursor()는 CWinApp() 클래스의 맴버 함수이다. 따라서 해당 함수를 호출하기 전에 AfxGetApp()를 먼저 호출하여 어플리케이션 객체의 포인터를 얻는 과정이 선행되어야 한다.

***참조**

맴버함수	기능
AfxGetMainWnd()	메인 프레임 클래스의 객체 포인터 획득
AfxGetApp()	어플리케이션 클래스의 객체 포인터 획득
AfxGetInstanceHandle()	응용 프로그램 인스턴스 핸들을 리턴
AfxGetAppName()	응용 프로그램의 이름을 리턴

4.2 애니메이션 커서 사용하기

에니메이션 커서를 사용하기 위해 메시지 핸들러 OnSetCursor() 를 다음과 같이 수정한다.

```
BOOL CTESTView::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)
{
	// TODO: Add your message handler code here and/or call default
	::SetCursor(AfxGetApp()->LoadCursor("c:₩₩Dog.ani"));
	return TRUE;
}
```

* 커서가 변경될 때 마다 호출되는 핸들러 함수이며, 다음과 같이 수정해서 확인해 보자.

```
void CTESTView::OnChangeCurosr()
{
	// TODO: Add your command handler code here
	HCURSOR hCursor;
	hCursor = AfxGetApp()->LoadCursor(IDC_CURSOR1);
	::SetClassLong(m_hWnd,GCL_HCURSOR,(long)hCursor);
}
```

```
void CTESTView::OnChangeCurosr()
{
	// TODO: Add your command handler code here
```

```
        HCURSOR hCursor;
        hCursor = ::LoadCursorFromFile("C:₩₩Dog.ani);
}
```

4.3 트레이 아이콘

윈도우 OS의 바탕화면에는 각종 작업을 간편하게 해주는 작업 표시줄이 있다. 작업 표시줄의 오른쪽 구석을 보면 사각형 영역이 있는데 이를 트레이라고 하며, 이 영역에 등록된 아이콘을 트레이 아이콘이라고 한다. 일반적으로 트레이 아이콘에 커서를 갖다 대면 툴팁이 출력되고 마우스 우측 버튼으로 클릭하면 간단한 팝업 메뉴가 출력된다.

1. 먼저 트레이에 등록될 아이콘을 리소스뷰에 IDI_TRAYICON로 설정한다.

2. 메뉴 핸들러 호출한다.

```
void CMainFrame::OnTry(void)
{
        ShowWindow(SW_HIDE);

        NOTIFYICONDATA nid;
        nid.cbSize = sizeof(nid);
        nid.hWnd = m_hWnd;
        nid.uID = IDI_ICON1;
        nid.uFlags = NIF_MESSAGE | NIF_ICON | NIF_TIP;
        nid.uCallbackMessage = WM_MY_TASKBAR;
        nid.hIcon = AfxGetApp()->LoadIcon(IDI_ICON1);

        TCHAR strTitle[256];
        GetWindowText(strTitle, sizeof(strTitle));
        lstrcpy(nid.szTip, strTitle);
        Shell_NotifyIcon(NIM_ADD, &nid);
        SendMessage(WM_SETICON, (WPARAM)TRUE, (LPARAM)nid.hIcon);
```

```
        m_bTray = TRUE;
}
```

```
class CMainFrame : public CFrameWnd
{
        // 맴버 변수 등록
        BOOL m_bTray;
```

```
CMainFrame::CMainFrame()
{
        // TODO: 여기에 멤버 초기화 코드를 추가합니다.
        m_bTray = FALSE;
}
```

```
#define WM_MY_TASKBAR WM_USER+10
class CMainFrame : public CFrameWnd
{
        //사용자 정의 메시지 함수
   LONG TrayIconSetting(WPARAM wParam, LPARAM lParam);
```

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        ON_MESSAGE(WM_MY_TASKBAR, TrayIconSetting)
        ON_WM_CREATE()
END_MESSAGE_MAP()
```

```
LONG CMainFrame:: TrayIconSetting(WPARAM wParam, LPARAM lParam) {
   if( lParam == WM_LBUTTONDOWN ) {
       AfxMessageBox(TEXT("TrayIcon을 선택했습니다."));
       ShowWindow(SW_SHOW);
   }
   return 0L;
}
```

```
void CMainFrame::OnDestroy()
{
        CFrameWnd::OnDestroy();
   if( m_bTray) {
```

```
        NOTIFYICONDATA nid;
        nid.cbSize = sizeof(nid);
        nid.hWnd = m_hWnd;
        nid.uID = IDI_ICON1;
        Shell_NotifyIcon(NIM_DELETE, &nid);
    }
}
```

예제 4.2 트레이 아이콘

4.4 사용자 메뉴의 생성 및 활용

step1) CBrush 를 이용하여 화면에 사각형을 출력해보자.

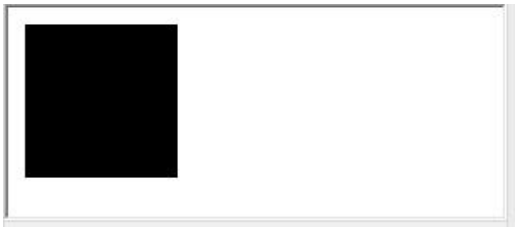
```
class C예제43메뉴View : public CView
{
private:
        COLORREF m_color; // 색을 나타내는 type
```

```
C예제43메뉴View::C예제43메뉴View()
{
        // TODO: 여기에 생성 코드를 추가합니다.
        m_color = RGB(0,0,0); // 검정으로 초기화.
}
```

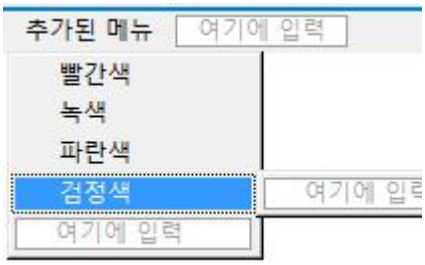
```
void C예제43메뉴View::OnDraw(CDC* pDC)
{
        C예제43메뉴Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
                return;
        // TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
        CBrush brush(m_color);
```

```
        CBrush* pOld = pDC->SelectObject( &brush ); // DC 의 brush 변경
        pDC->Rectangle( 10,10, 100, 100);
}
```

예제 4.3 메뉴1



step2) 메뉴를 추가하고 WM_COMMAND 처리를 하자.



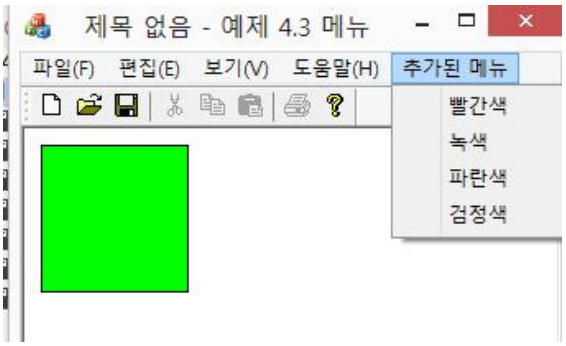

```
//resource.h
#define ID_COLOR_RED
 32775
#define ID_COLOR_GREEN
 32776
#define ID_COLOR_BLUE
 32777
#define ID_COLOR_BLACK
 32778
```

```
//메시지 핸들러 등록
BEGIN_MESSAGE_MAP(C예제43메뉴View, CView)
        // 표준 인쇄 명령입니다.
        ON_COMMAND_RANGE( ID_COLOR_RED, ID_COLOR_BLACK, OnColor )
END_MESSAGE_MAP()
```

```
// C예제43메뉴View 메시지 처리기
void C예제43메뉴View::OnColor(UINT nID)
```

```
{
        switch( nID ) // 메뉴 ID
        {
        case ID_COLOR_RED:
                m_color = RGB(255,0,0);             break;
        case ID_COLOR_GREEN:
                m_color = RGB(0,255,0);             break;
        case ID_COLOR_BLUE:
                m_color = RGB(0,0,255);             break;
        case ID_COLOR_BLACK:
                m_color = RGB(0,0,0);               break;
        }
        // 화면을 다시 그리게 한다.-> WM_PAINT 발생 -> OnPaint()호출.
        InvalidateRect(0);
}
```

예제 4.4 메뉴2



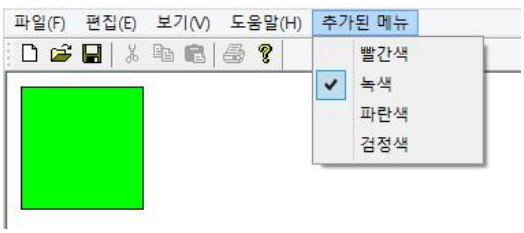
step3) 선택된 메뉴의 체크 표시를 하자.

```
class C예제43메뉴View : public CView
{
public:
        void OnUpdate(CCmdUI *pCmdUI);
```

```
BEGIN_MESSAGE_MAP(C예제43메뉴View, CView)
        // POPUP 메뉴가 화면에 나타날때 처리하기(check 넣기)
   ON_UPDATE_COMMAND_UI_RANGE( ID_COLOR_RED, ID_COLOR_BLACK,
                    OnUpdate )
END_MESSAGE_MAP()
```

```
// 메뉴의 Update 명령에 대한 함수 모양
void C예제43메뉴View::OnUpdate(CCmdUI *pCmdUI)
{
        switch( pCmdUI->m_nID )
        {
        case ID_COLOR_RED:
                pCmdUI->SetCheck( m_color == RGB(255,0,0)); break;
        case ID_COLOR_GREEN:
                pCmdUI->SetCheck( m_color == RGB(0,255,0)); break;
        case ID_COLOR_BLUE:
                pCmdUI->SetCheck( m_color == RGB(0,0,255)); break;
        case ID_COLOR_BLACK:
                pCmdUI->SetCheck( m_color == RGB(0,0,0));   break;
        }
}
```

예제 4.5 메뉴3

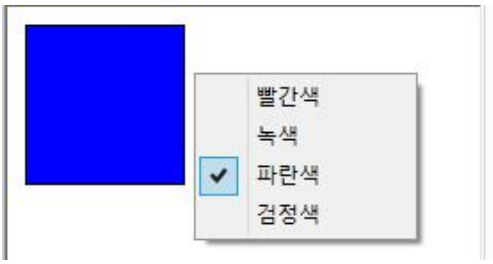


4.5 컨텍스트 메뉴

```
//클래스 마법사를 이용하여 WM_CONTECTMENU 추가
BEGIN_MESSAGE_MAP(C예제43메뉴View, CView)
        ON_WM_CONTEXTMENU()
END_MESSAGE_MAP()
```

```
void C예제43메뉴View::OnContextMenu(CWnd* /*pWnd*/, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가합니다.
        CMenu menu;
        menu.LoadMenu(IDR_MAINFRAME);
        CMenu *pMenu = menu.GetSubMenu(4);
        pMenu->TrackPopupMenu(TPM_LEFTALIGN|TPM_RIGHTALIGN,
                point.x, point.y, AfxGetMainWnd());
}
```

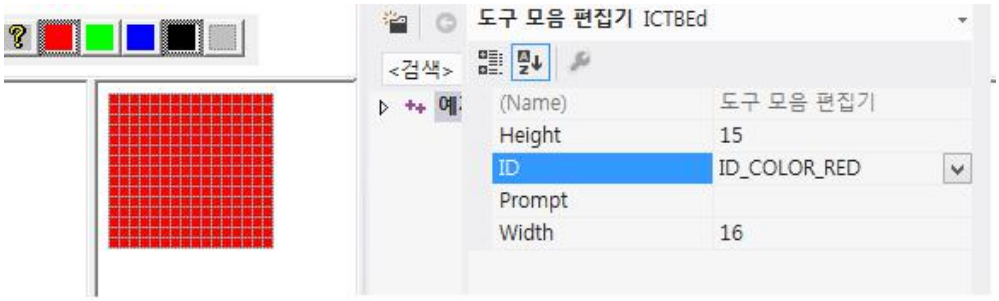
예제 4.6 컨텍스트 메뉴



4.6 툴바와 상태바

4.6.1 툴바

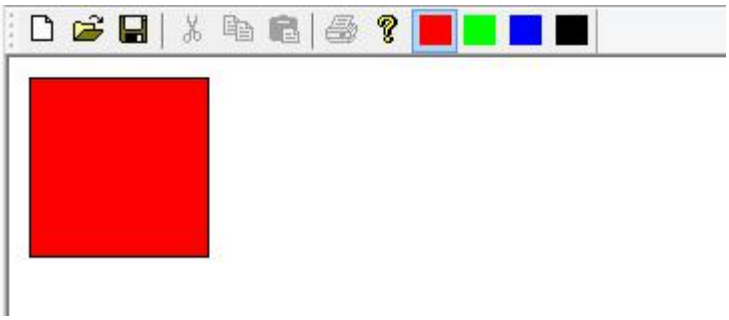
*** 먼저 아래와 같이 툴바 리소스를 등록한다.**



ID는 메뉴의 ID와 동일하게 설정한다.

```
//추가되는 코드는 없다.
```

예제 4.7 툴바



*** 툴바에 방향키 버튼을 추가해보자.**

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

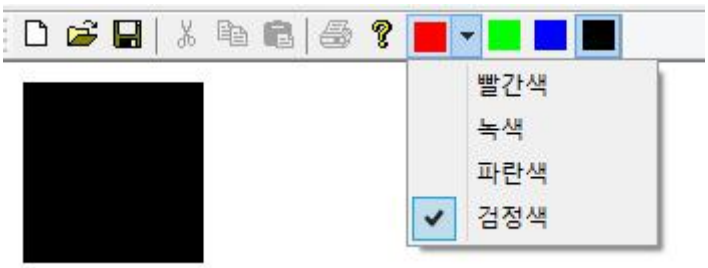
```
        ...
        // ToolBar의 고급기법 -> ToolBar가 내부적으로 가지고 있는
        // CToolBarCtrl의 멤버 함수 사용.
        CToolBarCtrl& tbc = m_wndToolBar.GetToolBarCtrl();

        // 단계 1. ToolBarCtrl style 변경.
        tbc.SetExtendedStyle( TBSTYLE_EX_DRAWDDARROWS );
        // 단계 2. 원하는 버튼에 화살표 넣기. ID->index
        int index = m_wndToolBar.CommandToIndex( ID_COLOR_RED );
        UINT style = m_wndToolBar.GetButtonStyle( index );
        style = style | TBSTYLE_DROPDOWN;
        m_wndToolBar.SetButtonStyle( index , style );
        ////////////////////////////////////////////////
```

```
// 메시지 핸들러 등록

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        ON_WM_CREATE()
        // ToolBar에서 화살표 버튼을 누를때 처리하기.
        ON_NOTIFY( TBN_DROPDOWN, AFX_IDW_TOOLBAR, &CMainFrame::OnDrop)
END_MESSAGE_MAP()
```

```
// CMainFrame 메시지 처리기
void CMainFrame::OnDrop(NMHDR *p1, LRESULT *ret)
{
        CRect r;
        NMTOOLBAR  *p = (NMTOOLBAR*)p1;
        m_wndToolBar.SendMessage( TB_GETRECT, p->iItem, // 버튼 ID
                                                        (LPARAM)&r);
        m_wndToolBar.ClientToScreen( &r);
        // Context 메뉴를 버튼 아래 나타낸다.
        CMenu menu;
        menu.LoadMenu( IDR_MAINFRAME );
```

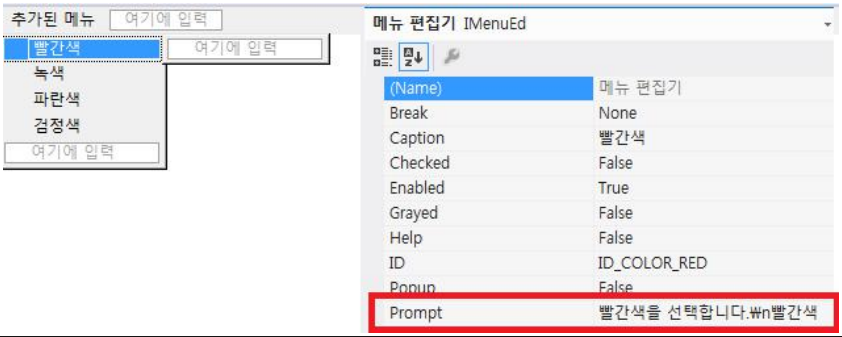
```
        CMenu* pSub = menu.GetSubMenu(4);
        pSub->TrackPopupMenu( TPM_LEFTBUTTON, r.left, r.bottom,       this);
}
```

예제 4.8 툴바 고급



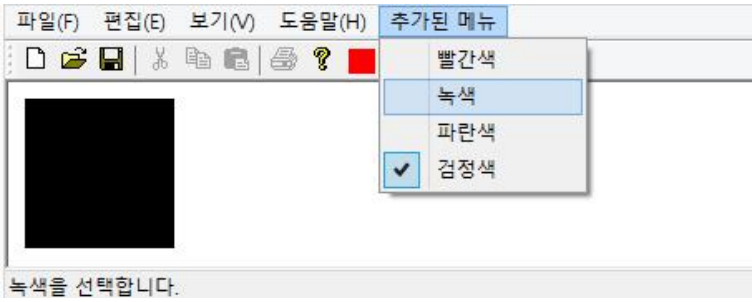
4.6.2 상태바

*** 상태바와 툴바에 메뉴의 정보를 출력해보자.**




```
// 상태바에 메뉴의 정보를 출력하고, 툴팁을 출력하는 내용임
// 소스상 수정 사항은 없고 메뉴 편집기에서 편집만 하면 됨
```

예제 4.9 상태바1



*** 툴바에 현재 시간을 출력해보자.**

문자열 편집기에 아래 내용을 추가한다.

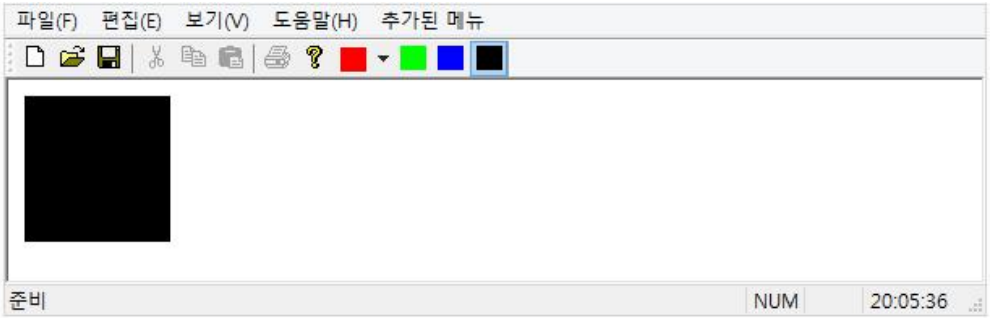

```
static UINT indicators[] =
{
        ID_SEPARATOR,           // 상태 줄 표시기
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
        ID_INDICATOR_TIME,    // 추가
};
```

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

```
{
        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;
        ////////////////////////////////////////////////
        // 상태바 Pane의 Style을 바꾼다. 결국 CStatusBar의 멤버 함수
        // 이용
        //m_wndStatusBar.SetPaneStyle( 4, SBPS_POPOUT);// 볼록
        m_wndStatusBar.SetPaneStyle( 4, SBPS_NOBORDERS);// 평면.

        // Timer 를 설정한다.
        SetTimer( 1, 1000, 0); // ID, 주기, 함수.
        // 처음에 1번 강제로 WM_TIMER를 보낸다.
        SendMessage( WM_TIMER, 1 ); // wParam = ID
```

```
void CMainFrame::OnTimer(UINT_PTR nIDEvent)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        //현재 시간을 상태바에 출력.
        if ( nIDEvent == 1 )
        {
                CTime t = CTime::GetCurrentTime(); // 현재 시간 얻기
                CString buf;
                buf.Format(TEXT("%02d:%02d:%02d"),
                        t.GetHour(), t.GetMinute(), t.GetSecond());
                m_wndStatusBar.SetPaneText(4, buf );
        }
        CFrameWnd::OnTimer(nIDEvent);
}
```

예제 4.10 상태바2



4.7 분할 윈도우

분할 윈도우는 여러 개의 Pane으로 구성된 윈도우로 동일한 데이터를 다른 방식으로 출력하거나 독립적이지만 서로 연관된 정보를 한눈에 볼 수 있게 한다.

각 Pane에는 다양한 형태의 뷰가 놓이게 되며, 분할바(Split Bar)를 이용하여 크기 조절이 가능하며 크게 동적 분할 윈도우, 정적 분할 윈도우 2개의 종류가 있다.

4.7.1 동적 분할 윈도우(Dynamic Splitter Window)

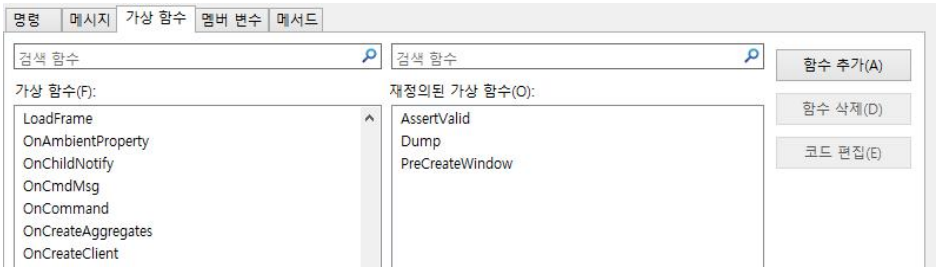
사용자가 필요할 때 각 페인으로 나눌 수 있는 분할 윈도우이다. VC++ 소스 코드 창이 여기에 해당된다

아래의 함수로 생성한다.

CSplitterWnd::Create()

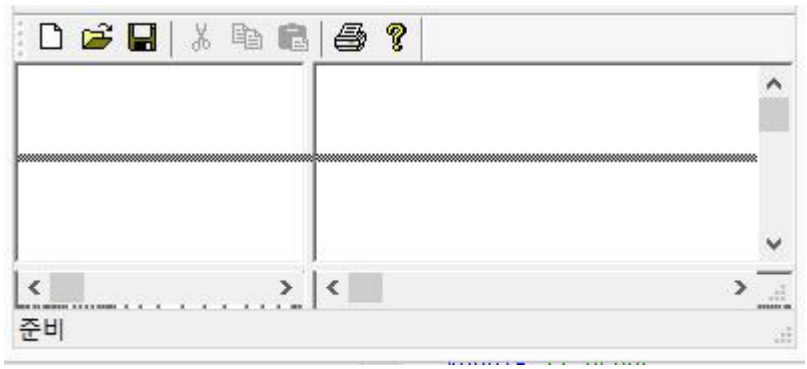
```
class CMainFrame : public CFrameWnd
{
protected:
        // 동적 Splitter 추가하기. 1단계
        CSplitterWnd m_wndSplitter;
```

// 가상함수 OnCreateClient 를 추가한다.




```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
        // TODO: 여기에 특수화된 코드를 추가 및/또는 기본 클래스를 호출합니다.
                // 동적 Splitter 만들기.
        return m_wndSplitter.Create( this, 2,2, CSize(100,100),pContext);
        // 동적 Splitter은 같은 종류의 view( CSampleView)만 가지고
        // 있기 때문에 사용자가 splitter만 만들면 View는 MFC가 자동으로
        // 만들어 준다.
        return CFrameWnd::OnCreateClient(lpcs, pContext);
}
```

예제 4.11 동적 분할 윈도우



4.7.2 정적 분할 윈도우(Static Splitter Window)

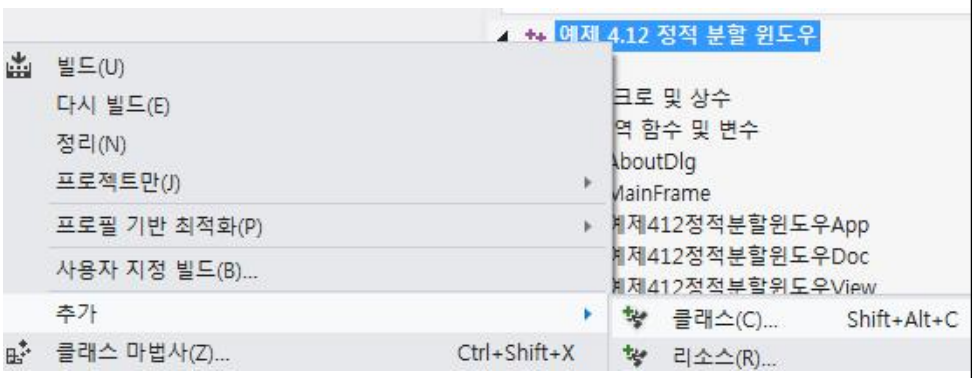
메인 윈도우가 생성될 때 페인의 수가 결정되어 변경할 수 없는 분할 윈도우이며, 분할 박스를 이용하여 페인의 크기를 조절한다.윈도우 탐색기가 대표적인 예이다.

아래의 함수를 이용하여 생성한다.

CSplitterWnd::CreateStatic()

정적 분할 윈도우는 서로 다른 뷰를 가진다.

// 뷰로 사용할 클래스를 추가한다.




```
// Doc 클래스에 선을 담을 컨테이너 변수를 선언한다.
struct Line
{
```

```
        CPoint ptFrom;
        CPoint ptTo;
        Line( CPoint p1, CPoint p2) : ptFrom(p1), ptTo(p2) {}
};

class C예제412정적분할윈도우Doc : public CDocument
{
// Attributes
public:
        CList<Line*, Line*>        m_list;
```

```
void CPaneView::OnDraw(CDC* pDC)
{
C예제412정적분할윈도우Doc* pDoc = (C예제412정적분할윈도우Doc*)GetDocument();
        // TODO: 여기에 그리기 코드를 추가합니다.
                // View를 새롭게 만들경우 GetDocument()를 다시 만들어 주는게
        // 좋다.
        // Doc의 list내용을 2배 확대해서 그린다.
        POSITION pos = pDoc->m_list.GetHeadPosition();
        CPen pen(PS_SOLID, 25, RGB(255, 0,0));
        CPen* old = pDC->SelectObject( &pen ); // 수정.
        while ( pos )
        {
                Line* pLine = pDoc->m_list.GetNext(pos);
                pDC->MoveTo( pLine->ptFrom.x * 2, pLine->ptFrom.y*2);
                pDC->LineTo( pLine->ptTo.x * 2, pLine->ptTo.y *2);
        }
        pDC->SelectObject( old );
}
```

```
class CMainFrame : public CFrameWnd
{
```

```
public:
        // 정적 Splitter 추가하기 1단계.
        CSplitterWnd m_wndSplitter;
        CSplitterWnd m_wndSplitter2;
```

```
#include "예제 4.12 정적 분할 윈도우View.h"
#include "PaneView.h"
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
        // TODO: 여기에 특수화된 코드를 추가 및/또는 기본 클래스를 호출합니다.
                // 정적 Splitter 만들기.
        // 1. splitter 를 먼저 만든다.
        m_wndSplitter.CreateStatic(this, 1, 2);
        // 2. 정적 splitter은 각View가 다를수 있으므로 직접 view를
        // 만들어 주어야 한다.
        m_wndSplitter.CreateView(0,0,                    RUNTIME_CLASS(C예
제412정적분할윈도우View), // 동적 생성
                        CSize(100,100), pContext);
        // 0,1 pane에는 splitter을 넣는다.
        m_wndSplitter2.CreateStatic( &m_wndSplitter , // 부모.. ???   2, 1,
WS_CHILD | WS_VISIBLE,
                                m_wndSplitter.IdFromRowCol(0,1) ); //
ID !!!
        // 새로운 splitter위에 View를 넣는다.
        m_wndSplitter2.CreateView( 0,0, RUNTIME_CLASS( CPaneView),
                                CSize(100,100), pContext);
        m_wndSplitter2.CreateView( 1,0, RUNTIME_CLASS( CPaneView),
                                CSize(100,100), pContext);
        return TRUE;
}
```

```
class C예제412정적분할윈도우View : public CView
{
```

```
        // Attributes
public:
        CPoint m_ptFrom;
        CPoint m_ptOld;
```

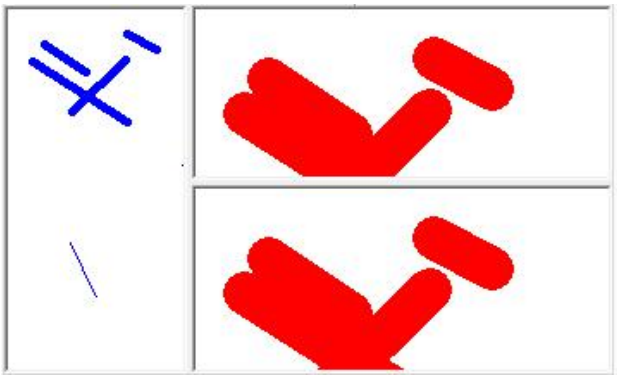
```
void C예제412정적분할윈도우View::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        m_ptFrom = m_ptOld = point;
                SetCapture();
        CView::OnLButtonDown(nFlags, point);
}

void C예제412정적분할윈도우View::OnLButtonUp(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        if ( GetCapture() == this )
        {
                ReleaseCapture();
                CClientDC dc(this);
                CPen pen( PS_SOLID, 5, RGB(0,0,255));
                CPen* old = dc.SelectObject(&pen);
                dc.MoveTo(m_ptFrom);
                dc.LineTo(point);
                dc.SelectObject(old);
                // Document에 저장한다.
                C예제412정적분할윈도우Doc* pDoc = GetDocument();
                pDoc->m_list.AddTail( new Line(m_ptFrom, point) );
                // 모든 View를 다시 그리게 한다.
                pDoc->UpdateAllViews(this);
        }
```

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

void C예제412정적분할윈도우View::OnMouseMove(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        if ( GetCapture() == this )
        {
                CClientDC dc(this);
                CPen pen( PS_SOLID, 5, RGB(0,0,255));
                CPen* old = dc.SelectObject(&pen);
                dc.SetROP2( R2_NOTXORPEN );
                dc.MoveTo(m_ptFrom);
                dc.LineTo(m_ptOld);
                dc.MoveTo(m_ptFrom);
                dc.LineTo(point);
                m_ptOld = point;
                dc.SelectObject(old);
        }
        CView::OnMouseMove(nFlags, point);
}
```

```
void C예제412정적분할윈도우View::OnDraw(CDC* pDC)
{
        C예제412정적분할윈도우Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)                      return;
        // TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
        CPen pen(PS_SOLID, 1, RGB(0,0,255));
        CPen* old = pDC->SelectObject(&pen);
        POSITION pos = pDoc->m_list.GetHeadPosition();
        while ( pos )
```

```
        {
                Line* p = pDoc->m_list.GetNext( pos );
                pDC->MoveTo( p->ptFrom);
                pDC->LineTo( p->ptTo);
        }
        pDC->SelectObject( old );
}
```

예제 4.12 정적 분할 윈도우



5. 사용자 인터페이스(2)

다이얼로그 박스(Dialog Box)는 응용 프로그램과 사용자간의 상호작용을 위해 가장 많이 사용되는 윈도우이다. 사용자와의 간편한 인터페이스가 중요시되는 현실에서 다이얼로그의 사용방식을 이해하는 것은 중요하다.

다이얼로그 박스는 크게 모달 다이얼로그(Modal Dialog) 와 모달리스 다이얼로그(Modeless Dialog)가 있다.

5.1 다이얼로그 및 컨트롤

다이얼로그 리소스 위에 배치된 각종 컨트롤은 사용자의 입력을 직접 처리하기 위한 중요한 구성 요소이다. 컨트롤은 크게 표준 컨트롤, 공통 컨트롤, 커스텀 컨트롤 세 가지로 구분할 수 있다.

5.1.1 표준 컨트롤(Standard control)

컨트롤	클래스명	기능
Picture	CStatic	비트맵, 아이콘 등 표시
Static Text	CStatic	정적 텍스트 출력
Edit Box	CEdit	텍스트 입력
Group Box	CButton	다른 컨트롤을 그룹화
Button	CButton	명령/기능 수행
Check Box	CButton	여러 개 선택 가능
Radio Button	CButton	하나만 선택 가능
List Box	CListBox	여러 항목을 표시
Combo Box	CBomboBox	데이터 입력과 목록화
Scroll Bar	CScrollBar	범위 내의 값 선택

```
// control ID
#define IDC_EDIT                1
```

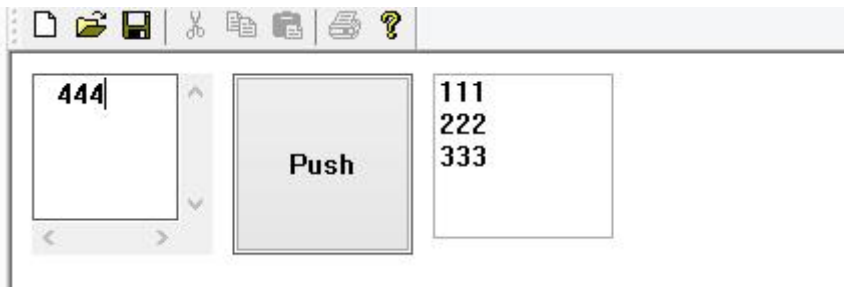
```
#define IDC_BUTTON        2
#define IDC_LIST                3

class CMy51표준컨트롤View : public CView
{
private:
        CEdit     m_edit;
        CButton  m_btn1;
        CListBox m_list;
```

```
int CMy51표준컨트롤View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
                return -1;

                // TODO:  여기에 특수화된 작성 코드를 추가합니다.
                // Control 을 생성한다.
                m_edit.Create(
                        WS_CHILD | WS_VISIBLE | WS_BORDER |
                        ES_MULTILINE | WS_HSCROLL | WS_VSCROLL,
                                                CRect(10,10, 100,100),
this, IDC_EDIT);
                m_btn1.Create( TEXT("Push"),
                                WS_CHILD | WS_VISIBLE | WS_BORDER |
BS_PUSHBUTTON,
                                CRect(110,10,200,100), this, IDC_BUTTON);
                m_list.Create( WS_CHILD | WS_VISIBLE | WS_BORDER ,
                                CRect(210,10,300,100), this, IDC_LIST);
                m_edit.SetMargins( 10, 10);
                return 0;
}
```

```
// 버튼의 핸들러 함수 메시지 맵 등록
BEGIN_MESSAGE_MAP(CMy51표준컨트롤View, CView)
        // 표준 인쇄 명령입니다.
        ON_WM_CREATE()
        // 자식 윈도우가 보내는 통지 처리하기.
        //ON_BN_CLICKED ( IDC_BUTTON, OnButton ) // 버튼일경우 한정.
        ON_CONTROL ( BN_CLICKED, IDC_BUTTON, OnButton) // 일반적인 표현.
END_MESSAGE_MAP()
```

```
void CMy51표준컨트롤View::OnButton()
{
        // edit에서 내용을 얻는다.
        CString msg;
        m_edit.GetWindowText( msg );

        // list에 넣는다.
        m_list.AddString( msg);
        // SendMessage( m_list.m_hWnd, LB_ADDSTRING, 0, (LPARAM)&msg);

        // edit를 지운다.
        m_edit.SetWindowText(TEXT(""));
}
```

5.1 표준 컨트롤



5.1.2 공통 컨트롤(Standard control)

윈도우즈 95부터 제공하는 컨트롤이다.

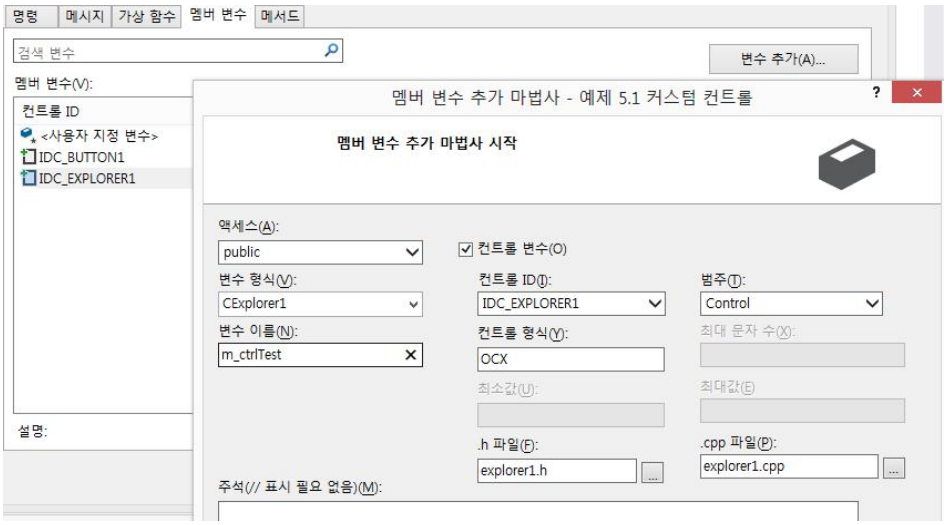
컨트롤	클래스명	기능
Spin	CSpinButtonCtrl	숫자값을 증가/감소
Progress	CProgreessCtrl	작업 진행 상태 표시
Slider	CSliderCtrl	스크롤 바와 유사
Hot Key	CHotKeyCtrl	키 조합 표시
List Control	CListCtrl	다양한 항목 표시
Tree Control	CTreeCtrl	계층적 항목 표시
Tab Control	CTabCtrl	여러 페이지 표시
Animate	CAnimateCtrl	AVI 파일 출력
Rich Edit	CRichEditCtrl	에디트 컨트롤의 확장

5.1.3 커스텀 컨트롤(Custom control)

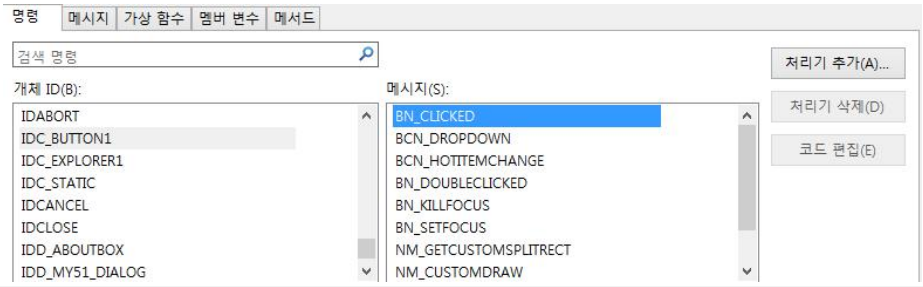
일반적으로 확장자가 ocx(ActiveX 컨트롤) 로 되어 있는 특별한 형태의 DLL이며, 자신의 시스템에 등록된 ActiveX 컨트롤 목록을 볼 수 있다.

//윈도우 창에서 마우스 오른쪽 버튼을 클릭한다.

// 클래스 마법사에서 아래의 흐름으로 사용할 컨트롤의 변수를 추가한다.



// 버튼을 하나 추가하고 핸들러 함수를 생성한다.




```
void C예제51커스텀컨트롤Dlg::OnBnClickedButton1()
{
        // TODO: 여기에 컨트롤 알림 처리기 코드를 추가합니다.
        m_ctrlTest.Navigate(TEXT("www.daum.net"), 0, 0, 0, 0);
}
```

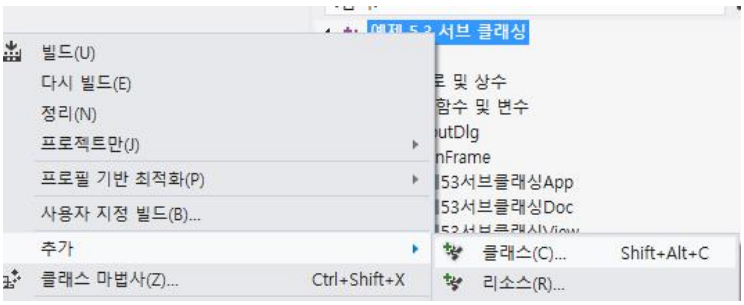
예제 5.2 커스텀 컨트롤

5.1.4 서브 클래싱

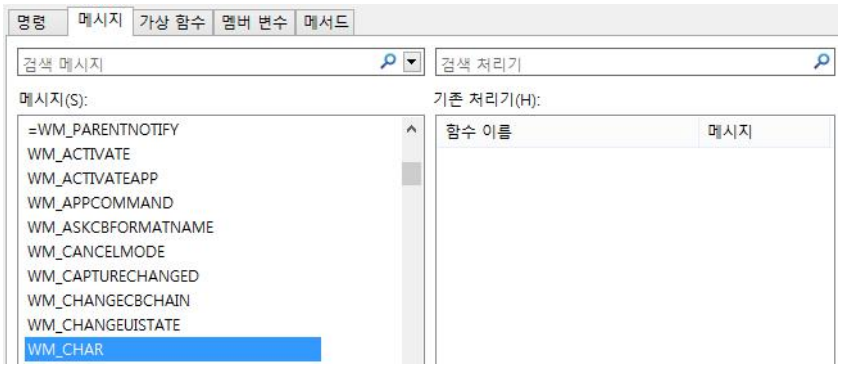
서브 클래싱이란 이미 생성되어 있는 특정 윈도우의 윈도우 프로시저 함수를 임의의 다른 함수로 대체하여 처리하는 기법을 말한다. 특정 메시지의 처리나 윈도우의 기능을 수정 할 목적으로 사용한다.

아래 예제는 숫자만 입력받는 에디트 컨트롤이다.

CEdit 를 상속받는 파생 클래스를 생성한다.



// WM_CHAR 이벤트 핸들러를 추가한다.



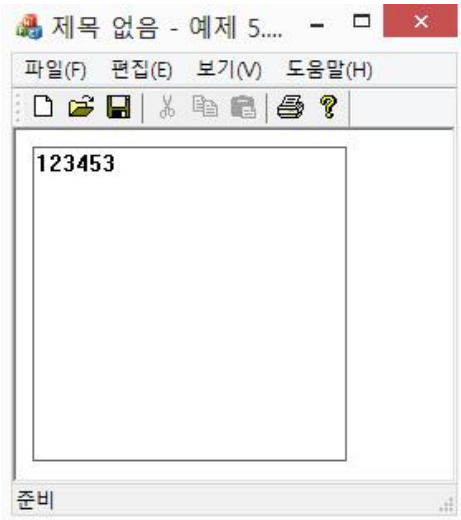

```
void CNumEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        // 숫자일경우만 부모에게 전달. 원래의 기능을 사용한다.

        if ( ( nChar >= '0' && nChar <= '9' ) || nChar == 8) // bs key
```

```
                CEdit::OnChar(nChar, nRepCnt, nFlags);

        // 그외의 문자는 아무 일도 하지 않는다.
        //CEdit::OnChar(nChar, nRepCnt, nFlags);
}
```

```
#include "NumEdit.h" // CNumEdit class 의 헤더 파일.

#define IDC_EDIT 1

class C예제53서브클래싱View : public CView
{
        CNumEdit m_edit;
```

```
int C예제53서브클래싱View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
                return -1;

                // TODO:  여기에 특수화된 작성 코드를 추가합니다.
                // Edit Box 만들기.
                m_edit.Create( WS_CHILD | WS_VISIBLE | WS_BORDER,
                CRect(10,10, 200, 200), this, IDC_EDIT); // id
                return 0;
}
```

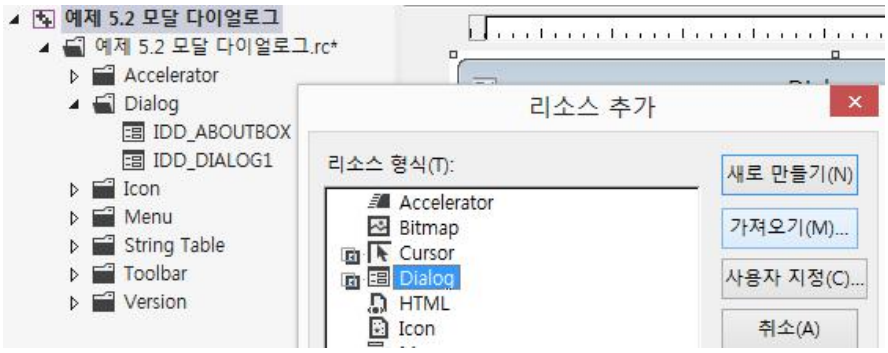
예제 5.3 서브 클래싱



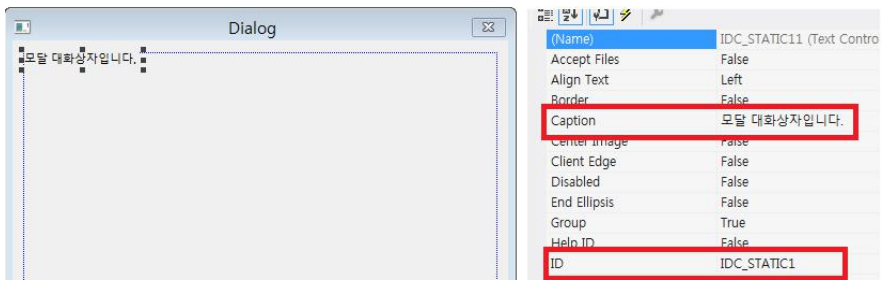
5.2 모달 다이얼로그

다이얼로그가 종료되기 전까지 다이얼로그를 호출한 윈도우로 작업 전환이 되지 않는 다이얼로그가 있다. 이를 모달 다이얼로그라 한다.

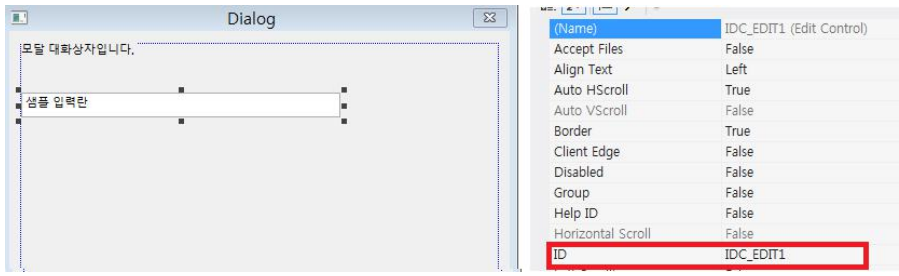
// 모달 다이얼로그로 사용될 다이얼로그를 생성한다.



// 생성된 다이얼로그에 static 컨트롤을 추가한다.

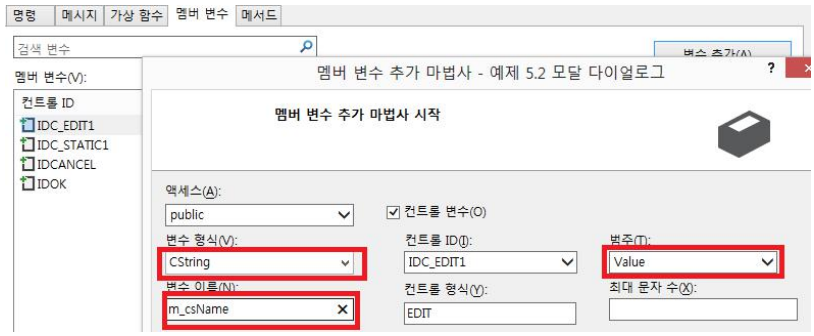


// 에디트 컨트롤을 추가한다.



// 다이얼로그를 더블 클릭하면 해당 다이얼로그를 관리하는 클래스를 생성할 수 있다.

// Edit 컨트롤의 맴버 변수를 추가한다.



// 아래 맴버 함수를 추가한다.

```
void CTestDlg::SetMessage(CString str)
{
	m_csName = str;
}
```

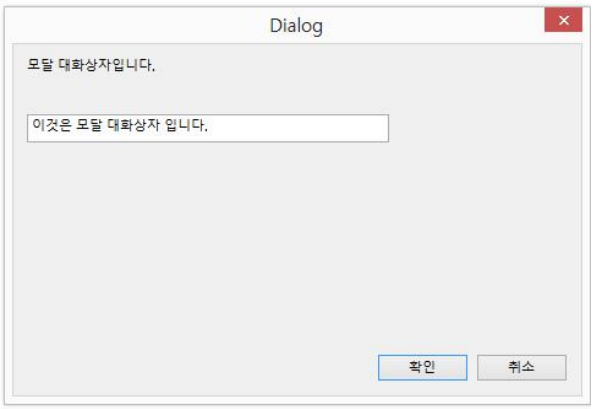
```
// 생성된 CTestDlg 클래스에 OK 버튼의 핸들러를 생성한 후 아래 코드를 추가한다.
void CTestDlg::OnBnClickedOk()
{
	// TODO: 여기에 컨트롤 알림 처리기 코드를 추가합니다.
	UpdateData(TRUE);
	if(m_csName.GetLength() == 0)
	{
		CEdit *pEdit = (CEdit*)GetDlgItem(IDC_EDIT1);
		AfxMessageBox(TEXT("이름을 입력하세요"));
		pEdit->SetFocus();
		return;
	}
	CDialogEx::OnOK();
}
```

```
//모달 다이얼로그를 호출한다.
#include "TestDlg.h"
void C예제52모달다이얼로그View::OnLButtonDown(UINT nFlags, CPoint point)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
	CTestDlg Dlg;
	Dlg.SetMessage(TEXT("이것은 모달 대화상자 입니다."));
	if(Dlg.DoModal() == IDOK)
	{
		CString temp = _T("");
		temp.Format(TEXT("입력한 이름은 %s입니다."), Dlg.m_csName);
		AfxMessageBox(temp);
	}
```

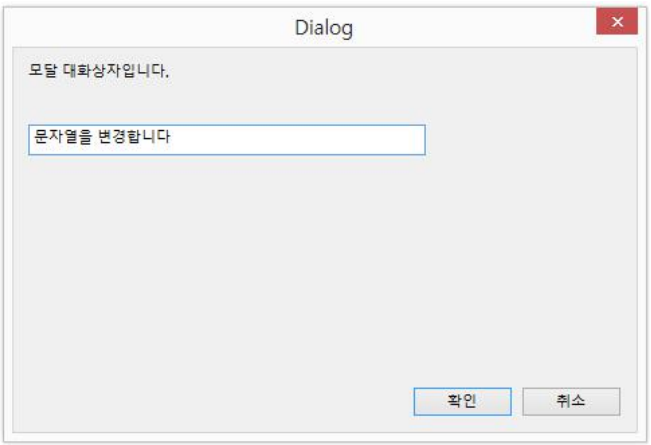
```
        else
                AfxMessageBox(TEXT("취소 되었음"));
        CView::OnLButtonDown(nFlags, point);
}
```

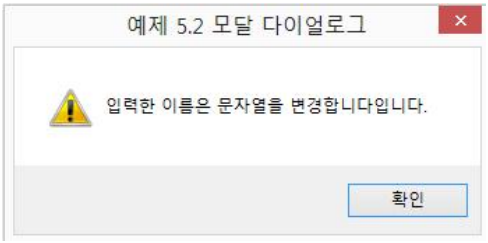
예제 5.4 모달 다이얼로그

*** 마우스 왼쪽 버튼 클릭시**



*** 문자열 변경 후 확인 버튼 클릭시**





5.3 모달리스 다이얼로그

모달리스 다이얼로그는 모달 다이얼로그와는 달리 다이얼로그를 종료하지 않더라도 프로그램에서 다른 작업을 할 수 있어야 한다. 이런 이유로 다이얼로그가 종료되면 리턴하는 DoModal() 함수는 모달리스 다이얼로그에 부적합하다.

모달 다이얼로그와 모달리스 다이얼로그를 만들 때 차이점이다.

구분	모달 다이얼로그	모달리스 다이얼로그
사용되는 생성자	다이얼로그 리소스 ID를 인자로 하는 생성자	인자없는 디폴트 생성자
윈도우를 생성하기 위해 사용하는 함수	DoModal()	Create()
생성되는 객체	일반적으로 스택에 다이얼로그객체를 생성	new 연산자를 이용하여 힙 메모리에 객체를 생성
다이얼로그 종료	EndDialog()	DestroyWindow()

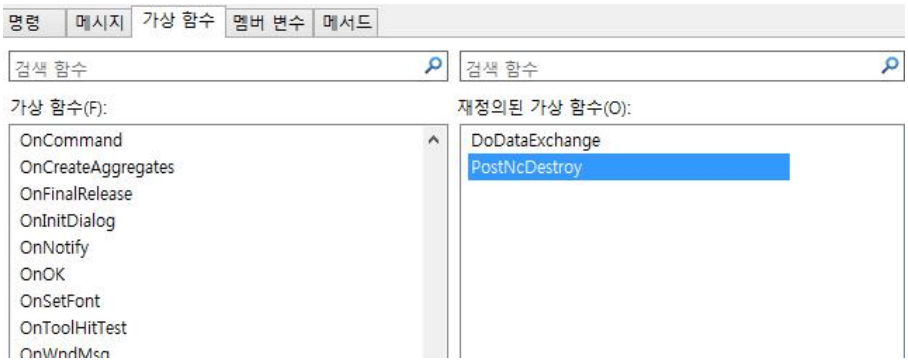
모달리스 다이얼로그 리소스 및 관련 클래스는 이전 예제에 생성했던 것을 사용할 것이다.

```
class CTestDlg : public CDialogEx
{
        DECLARE_DYNAMIC(CTestDlg)

public:
```

```
                    // 모달리스 관련 변수
          BOOL m_bSelfDelete;
```

```
CTestDlg::CTestDlg(CWnd* pParent /*=NULL*/)
          : CDialogEx(CTestDlg::IDD, pParent)
          , m_csName(_T(""))
{
                    m_bSelfDelete               = FALSE;
}
```

//가상함수 추가



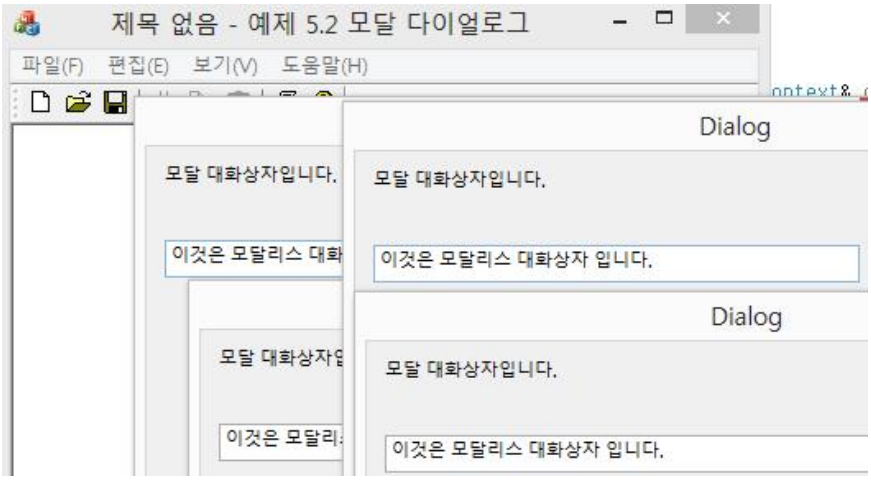

```
// WM_DESTROY --> OnDestroy --> PostNCDestroy()
// Non-Client 영역 파괴 : delet this 허용(다른 곳은 자기자신 파괴가 안된다.)
// 리턴되면서 클라이언트 뷰가 완전히 파괴된다.
void CTestDlg::PostNcDestroy()
{
          // TODO: 여기에 특수화된 코드를 추가 및/또는 기본 클래스를 호출합니다.
          if(m_bSelfDelete)
          {
                    AfxMessageBox(TEXT("test"));
                    delete this;
          }
          else
          {
```

```
                CDialog::PostNcDestroy();
        }
        CDialogEx::PostNcDestroy();
}
```

```
#include "TestDlg.h"
void C예제52모달다이얼로그View::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        // TODO: Add your control notification handler code here
        CTestDlg* pDlg = new CTestDlg;
        ASSERT(pDlg);

        pDlg->SetMessage(TEXT("이것은 모달리스 대화상자 입니다."));
        pDlg->Create(IDD_DIALOG1);
        pDlg->ShowWindow(SW_SHOW);



        // 모달리스가 여러개 생성되었을때 안정적인 종료 처리 방법
        // 객체가 자신을 delete 하게 됨
        pDlg->m_bSelfDelete = TRUE;
        CView::OnLButtonDown(nFlags, point);
}
```

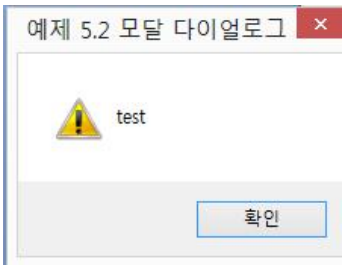
예제 5.5 모달리스 다이얼로그1

*** 마우스 왼쪽 버튼을 4번 클릭한 실행 화면**



*** 메인 윈도우 종료시 화면**

아래와 같은 메시지 박스 함수가 4번 호출된다.



위의 예제를 한 번만 실행 가능한 모달리스 다이얼로그 형태로 구성하였다.

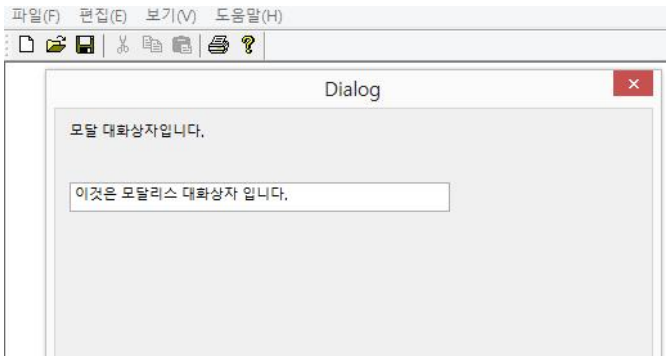
```
#include "TestDlg.h"


class C예제52모달다이얼로그View : public CView
{
        CTestDlg* m_pMDlg;                  // 1. 모달리스 선언
```

```
C예제52모달다이얼로그View::C예제52모달다이얼로그View()
{
        // TODO: 여기에 생성 코드를 추가합니다.
```

```
	// 2. 초기화(모달리스)
	m_pMDlg = NULL;
}
```

```
#include "TestDlg.h"
void C예제52모달다이얼로그View::OnLButtonDown(UINT nFlags, CPoint point)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
	if(m_pMDlg == NULL)
	{
		// 대화상자를 생성하지 않았으므로 메모리 할당하고 모달리스 생
성
		m_pMDlg = new CTestDlg;
		m_pMDlg->SetMessage(TEXT("이것은 모달리스 대화상자 입니
다."));
		m_pMDlg->Create(IDD_DIALOG1);
		// 윈도우가 제대로 생성되었다면 화면에 보인다.
		if(m_pMDlg->GetSafeHwnd())
			m_pMDlg->ShowWindow(SW_SHOW);
	}
	else
	{
		// 대화상자가 이미 생성되었다면 화면에 보여주고 focus이동한
다.
		m_pMDlg->ShowWindow(SW_SHOW);
		m_pMDlg->SetFocus();
	}
	// 모달리스의 종료
	// WM_DESTROY --> OnDestroy --> PostNCDestroy()
	CView::OnLButtonDown(nFlags, point);
}
```

예제 5.6 모달리스 다이얼로그2



5.4 공통 다이얼로그

메모장을 실행하여 [파일->열기...] 메뉴를 선택해 보자.
출력된 열기 다이얼로그는 메모장 뿐만 아니라 대부분의 응용 프로그램이 공통적으로 갖고 있다. 열기 다이얼로그는 대부분의 프로그램이 필요로 할 만큼 사용 빈도가 높다. 이처럼 활용 빈도가 높고 응용 프로그램에서 보편적으로 사용되는 다이얼로그는 윈도우즈 OS 가 서비스 차원에서 제공한다.

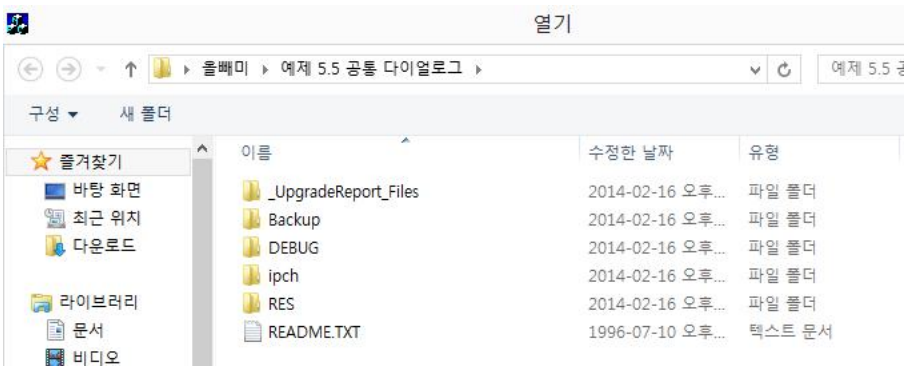
이를 공통 다이얼로그라고 한다. 공통 다이얼로그는 윈도우즈가 내장(comdlg32.dll)하고 있기 때문에 프로그래머가 직접 구현하지 않아도 프로그램에서 손쉽게 사용할 수 있다.

공통 다이얼로그에는 다음과 같은 종류가 있다.

	다이얼로그 유형	클래스명
모달 다이얼로그	파일열기 다른이름으로 저장	CFileDialog
	색상	CColorDialog
	폰트	CFontDialog
	인쇄 프린터 설정	CPrintDialog
	페이지 설정	CPageSetupDialog

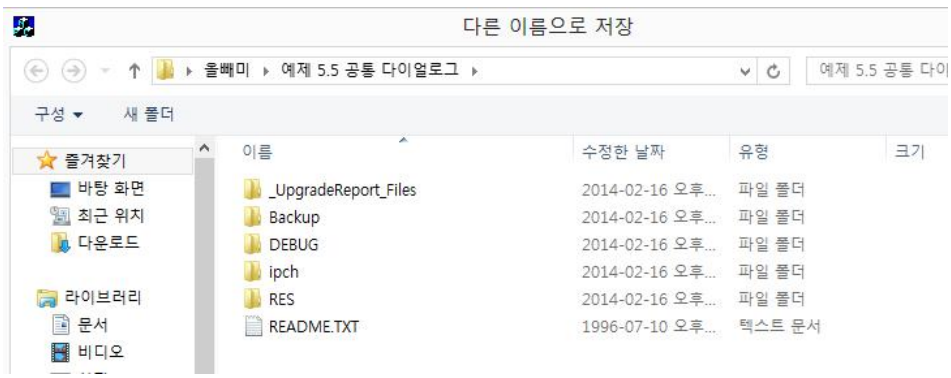
모달리스 다이얼로그	찾기 바꾸기	CFindReplaceDialog

//열기 다이얼로그




```
void CComDlgView::OnComdlgOpen()
{
	// TODO: Add your command handler code here
	CFileDialog dlg(TRUE, "txt", "*.txt");
	if(dlg.DoModal() == IDOK)
		AfxMessageBox(dlg.GetPathName());
}
```

//다른이름으로 저장 다이얼로그



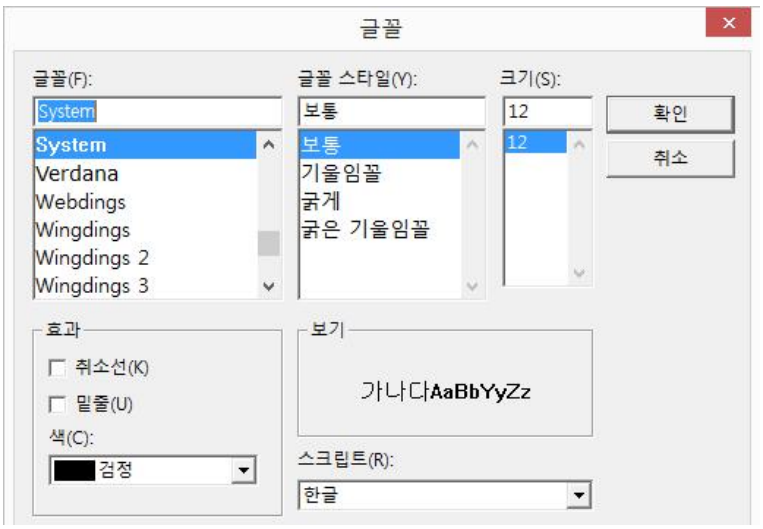
```
void CComDlgView::OnComdlgSaveas()
{
	// TODO: Add your command handler code here
	CFileDialog dlg(FALSE, "txt", "*.txt");
	if(dlg.DoModal() == IDOK)
		AfxMessageBox(dlg.GetPathName());
}
```

```
// 색상 다이얼로그
```


```
void CComDlgView::OnComdlgColor()
{
	// TODO: Add your command handler code here
	COLORREF color = RGB(0, 0, 0);
	CColorDialog dlg(color);
	if(dlg.DoModal() == IDOK) {
		CString str;
		color = dlg.GetColor();
```

```
                str.Format("색상 : (%d, %d, %d)", GetRValue(color),

                  GetGValue(color),

                  GetBValue(color));
                AfxMessageBox(str);
        }
}
```

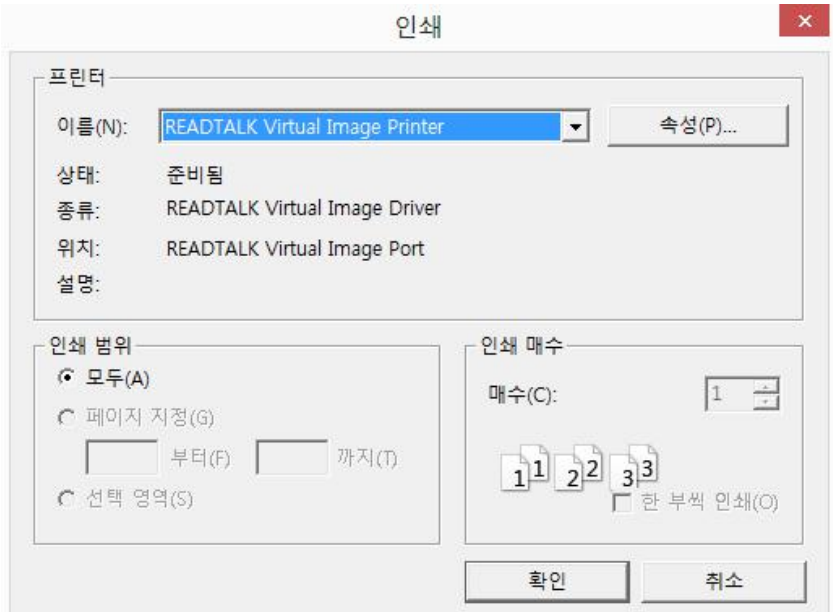
```
// 글꼴 다이얼로그
```




```
void CComDlgView::OnComdlgFont()
{
        // TODO: Add your command handler code here
        CClientDC dc(this);
        CFont* pFont = dc.GetCurrentFont();
        LOGFONT logFont;
        pFont->GetLogFont(&logFont);

        CFontDialog dlg(&logFont);
        if(dlg.DoModal() == IDOK) {
```

```
                CString str;
                str.Format("서체 : %s, 크기 : %d", dlg.GetFaceName(),
                   dlg.GetSize()/10);
                AfxMessageBox(str);
        }
}
```

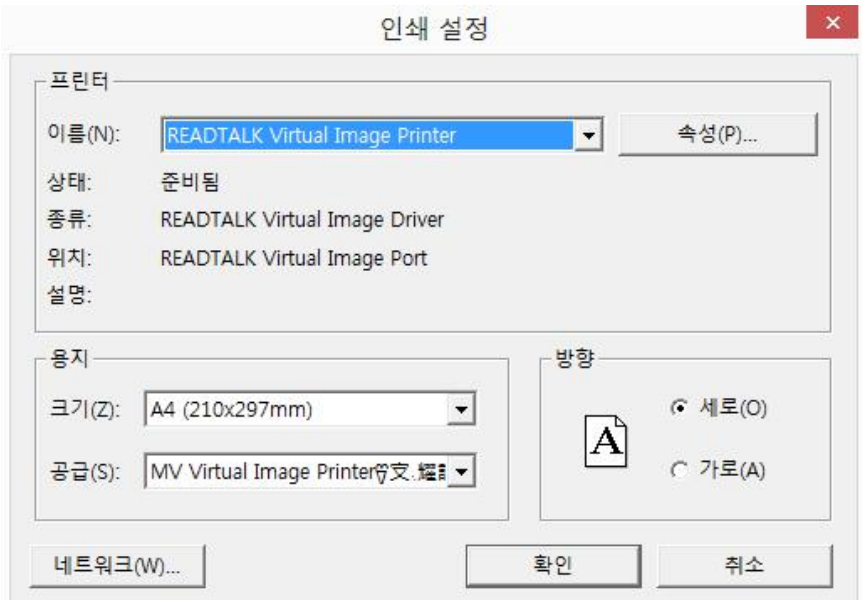
```
// 인쇄 다이얼로그
```




```
void CComDlgView::OnComdlgPrint()
{
        // TODO: Add your command handler code here
        CPrintDialog dlg(FALSE);
        if(dlg.DoModal() == IDOK) {
                CString str, str1;
                str1.Format("페이지 : %d - %d ₩n", dlg.GetFromPage(),
                        dlg.GetToPage());       str += str1;
                str1.Format("모든 페이지? : %s₩n",
```

```
                        dlg.PrintAll() ? "예" : "아니오");
                str += str1;
                str1.Format("선택 페이지? : %s₩n",
                        dlg.PrintSelection() ? "예" : "아니오");
                str += str1;
                str1.Format("범위 페이지? : %s₩n",
                        dlg.PrintRange() ? "예" : "아니오");
                str += str1;
                str1.Format("인쇄 매수 : %d", dlg.GetCopies());
                str += str1;
                AfxMessageBox(str);
        }
}
```

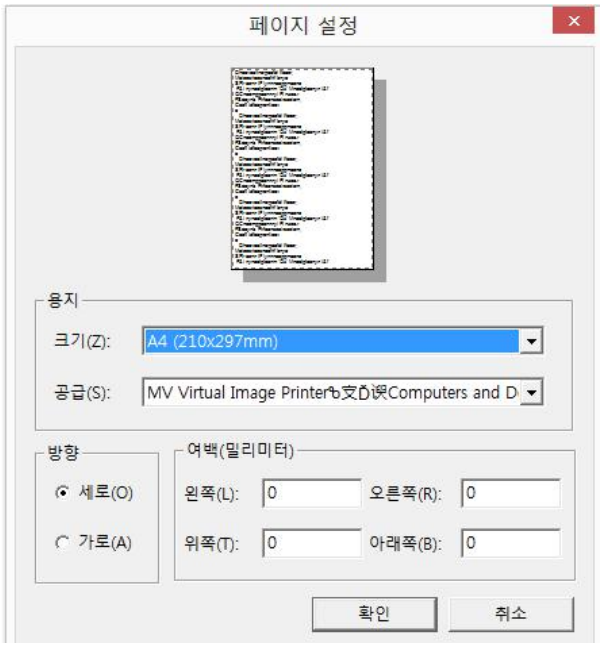
```
// 프린터 설정 다이얼로그
```




```
void CComDlgView::OnComdlgPrintersetup()
{
        // TODO: Add your command handler code here
```

```
        CPrintDialog dlg(TRUE);
        if(dlg.DoModal() == IDOK) {
                CString str;
                str.Format("%s, %s, %s", dlg.GetDeviceName(),

dlg.GetDriverName(),

dlg.GetPortName());
                AfxMessageBox(str);
        }
}
```

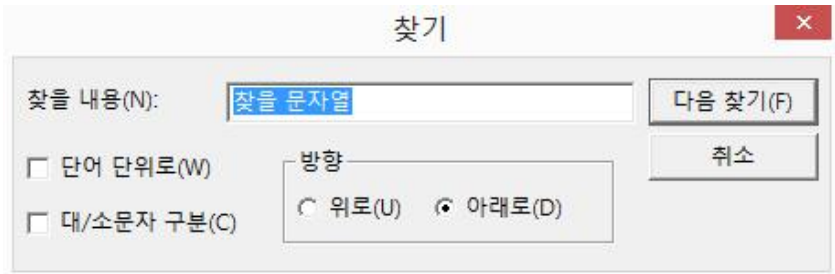
```
// 페이지 설정 다이얼로그
```




```
void CComDlgView::OnComdlgPagesetup()
{
        // TODO: Add your command handler code here
```

```
        CPageSetupDialog dlg;
        if(dlg.DoModal() == IDOK) {
                CString str, str1;
                CRect rectMargin;
                CSize sizePaper;
                dlg.GetMargins(&rectMargin, NULL);
                str1.Format("프린터 여백 : (%d, %d, %d, %d)₩n",
                                        rectMargin.left, rectMargin.top,
                                        rectMargin.right,
rectMargin.bottom);
                str += str1;
                sizePaper = dlg.GetPaperSize();
                str1.Format("용지 크기 : (%d, %d)", sizePaper.cx,

                        sizePaper.cy);
                str += str1;
                AfxMessageBox(str);
        }
}
```

```
// 찾기 다이얼로그
```




```
void CComDlgView::OnComdlgFind()
{
        // TODO: Add your command handler code here
        if(!pDlg) {
```

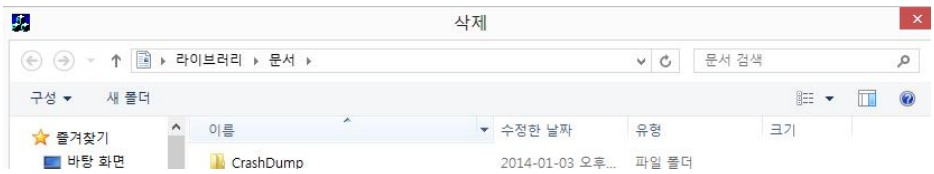
```
                pDlg = new CFindReplaceDialog;
                pDlg->Create(TRUE, "찾을 문자열", NULL, FR_DOWN, this);
        }
}
```

```
// 바꾸기 다이얼로그
```


```
LRESULT CComDlgView::OnFindReplaceMsg(UINT wParam, LONG lParam)
{
        CFindReplaceDialog* pDialog = CFindReplaceDialog::GetNotifier(lParam);
        CString str, str1;

        if(pDialog->IsTerminating()) {
                str = "다이얼로그 박스 종료";
                AfxMessageBox(str);
                pDlg = NULL;
                return TRUE;
        }

        str1.Format("찾을 문자열 : %s₩n", pDialog->GetFindString());
        str += str1;
        str1.Format("바꿀 문자열 : %s₩n", pDialog->GetReplaceString());
        str += str1;
```

```
        if(pDialog->FindNext())
                str1 = "다음 찾기";
        if(pDialog->ReplaceCurrent())               str1 = "바꾸기";
        if(pDialog->ReplaceAll())          str1 = "모두 바꾸기";
        str += str1;
        AfxMessageBox(str);
        return TRUE;
}
```

```
// 파일 삭제 다이얼로그
```




```
void CComDlgView::OnComdlgDelete()
{
        // TODO: Add your command handler code here
        CDeleteDialog dlg(TRUE, "obj", "*.obj");
        BOOL bRet = dlg.DoModal();
        CString msg;
        if(bRet == IDCANCEL && dlg.m_bDeleteAll) {
                msg = "*.obj 모든 파일을 삭제합니다.";
                AfxMessageBox(msg);
        }
        else if(bRet == IDOK) {
                msg.Format("%s 파일을 삭제합니다.", dlg.GetPathName());
                AfxMessageBox(msg);
        }
}
```

예제 5.7 공통 다이얼로그

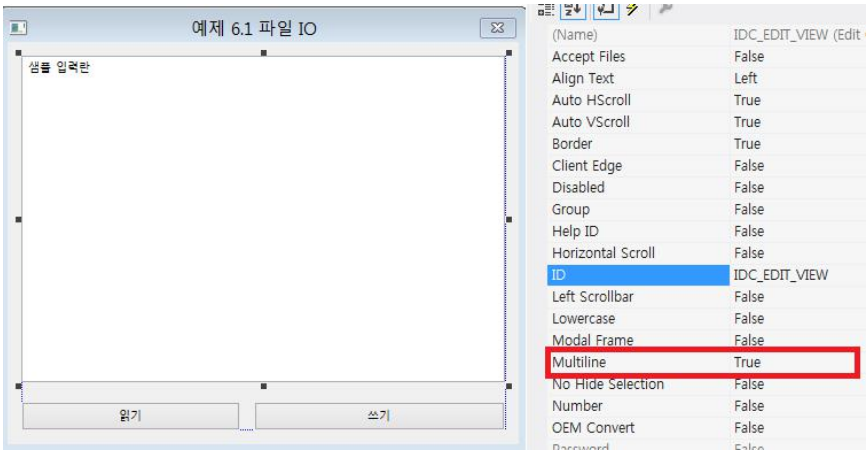
6. 파일 IO

6.1 일반적인 파일 입출력

먼저 디스크에 저장된 파일을 읽고/쓰기 위한 일반적인 방법을 살펴보자. MFC는 파일을 제어하기 위해 CFile 클래스를 제공한다. CFile 클래스의 주요 맴버함수는 다음과 같다.

맴버함수	내용
Open()	파일을 오픈한다 CFile 생성자에서도 파일을 오픈할 수 있다.
Close()	오픈된 파일을 닫는다.
Read()	오픈된 파일의 현재 위치에서 데이터를 읽어들인다.
Write()	오픈된 파일의 현재 위치에 데이터를 쓴다.
ReadHuge()	Read()와 동일한 기능을 수행하며 64KB 이상의 데이터를 읽어 들인다.
WriteHuge(0	Write()와 동일한 기능을 수행하며 64KB 이상의 데이터를 쓴다.
Seek()	오픈된 파일의 현재 위치를 변경한다.
GetLength()	오픈된 파일의 길이를 리턴한다.(바이트)
GetStatus()	오픈된 파일의 속성, 날짜 등의 상태를 가져온다.
GetFileName()	오픈된 파일의 이름을 리턴한다.

```
// 아래 그림처럼 컨트롤을 구성한다.
```




```
void C예제61파일IODlg::OnBnClickedBtnRead()
{
	// TODO: 여기에 컨트롤 알림 처리기 코드를 추가합니다.
	CFileDialog dlg(TRUE); // 열기 다이얼로그 객체
	if( dlg.DoModal() == IDOK)
	{
		CFile file;
		TCHAR str[50];
		wcscpy(str, dlg.GetFileName());
		file.Open(str, CFile::modeRead);//선택된 파일 오픈
		UINT nCount = file.GetLength() -1;  // 파일 길이

		TCHAR *buffer = new TCHAR[nCount];  //동적 메모리 할당
		ZeroMemory(buffer, nCount);
		file.Read(buffer, nCount); // 파일 읽기
		file.Close();                                  // 오픈된 파일 닫
음
		SetDlgItemText(IDC_EDIT_VIEW, buffer);//데이터 출력
		delete [] buffer;          // 힙 메모리 삭제
	}
}
```

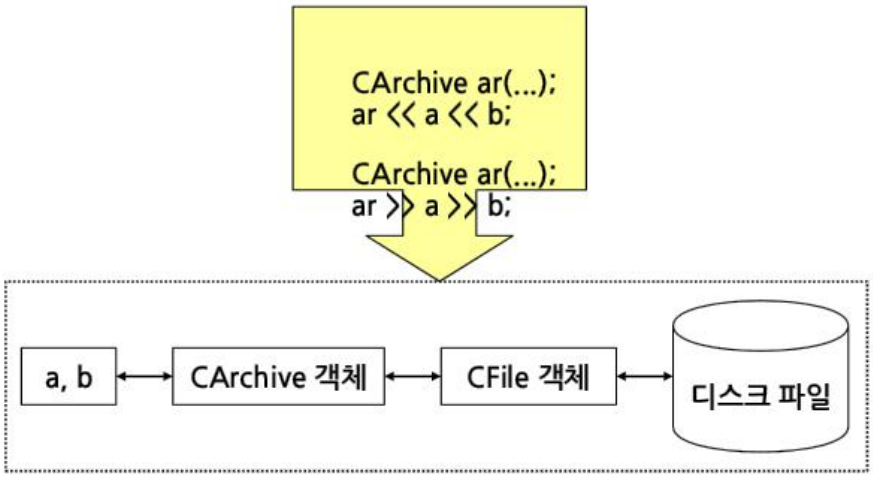
```
void C예제61파일IODlg::OnBnClickedBtnWrite()
{
	// TODO: 여기에 컨트롤 알림 처리기 코드를 추가합니다.
	CFileDialog dlg(FALSE); //저장 다이얼로그
	if( dlg.DoModal() == IDOK)
	{
		CFile file;
		file.Open(dlg.GetFileName(),
CFile::modeCreate||CFile::modeWrite);
		CString buffer;
		GetDlgItemText(IDC_EDIT_VIEW, buffer);
		file.Write((LPCSTR)(LPCTSTR)buffer, buffer.GetLength());
		file.Close();
		SetDlgItemText(IDC_EDIT_VIEW, TEXT(""));
	}
}
```

예제 6.1 파일 IO

6.2 직렬화(Serialization)

텍스트나 이미지와 같은 바이트 정보는 CFile 클래스만 이용해도 파일에 읽고 쓰기가 가능하다. 하지만 CRect 나 자신이 만든 클래스의 객체를 파일로 저장하려면 어떻게 해야 할까?

CRect 의 경우 맴버 변수 left, top, right, bottom 을 구분하여 일일이 저장하면 될 것이다. 하지만 CRect 처럼 클래스가 갖고 있는 모든 맴버변수를 구분하여 저장한다면 프로그래머에게 있어 번거로운 작업이 아닐 수 없다. 그래서 MFC는 이런 과정을 대신하는 가상함수 CObject::Serialize() 함수를 제공한다.



이 처럼 객체를 디스크와 같은 저장 장치에 저장하고 로드하는 일련의 처리과정을 직렬화라고 한다.

직렬화는 객체가 순차적으로 저장되고 로드되기 때문에 파일에 저장된 객체들 중 원하는 객체를 랜덤하게 접근할 수 없다. 이런 이유로 직렬화를 데이터베이스 대체 수단으로 사용할 수는 없다.

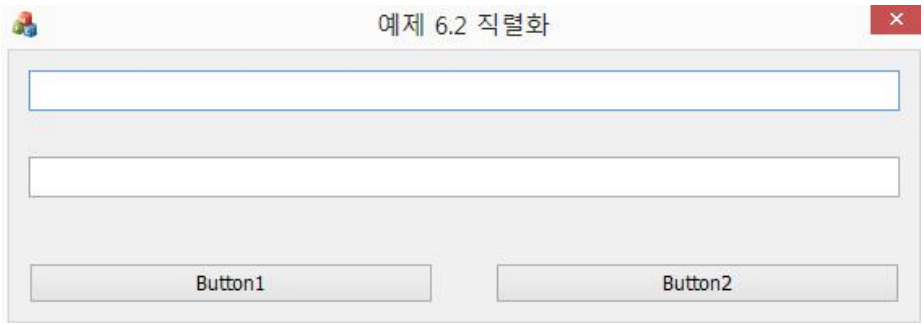
모든 클래스가 직렬화될 수 있는 것은 아니고 다음의 조건을 갖춘 클래스만이 직렬화가 가능하다.

1) CObject 로부터 파생되어야 한다.
2) 디폴트 생성자를 갖고 있어야 한다. 이 생성자는 파일에서 객체가 복원될 때 사용된다.
3) 클래스 정의 부분에 DECLARE_SERIAL() 매크로를 사용하고, 클래스 구현 파일에 IMPLEMENT)SERIAL() 매크로를 사용한다.
4) 가상함수 Serialize()을 오버라이드 해야 한다.

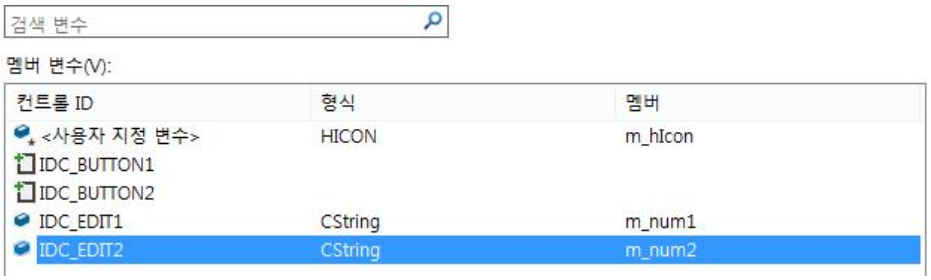
6.2.1 간단한 직렬화 구현해 보기

2개의 문자열을 직열화 하는 예제

// UI 구성



// 맴버 변수 추가




```
void C예제62직렬화Dlg::OnBnClickedButton1()
{
	// TODO: 여기에 컨트롤 알림 처리기 코드를 추가합니다.
	UpdateData();
	CFile f;
		f.Open(TEXT("C:₩₩test.txt"),
CFile::modeCreate|CFile::modeWrite);
		CArchive ar(&f, CArchive::store);
		ar << m_num1 << m_num2;	ar.Close();
	f.Close();
	m_num1 = "";
	m_num2 = "";
```

```
        UpdateData(TRUE);
}

void C예제62직렬화Dlg::OnBnClickedButton2()
{
        // TODO: 여기에 컨트롤 알림 처리기 코드를 추가합니다.
        CFile f;
        if(!f.Open(TEXT("c:₩₩test.txt"), CFile::modeRead))
                return;
               CArchive ar(&f, CArchive::load);
              ar >> m_num1 >> m_num2;
        ar.Close();
                f.Close();
        UpdateData(FALSE);
}
```

예제 6.2 직렬화

6.2.2 구조체 직렬화 하기

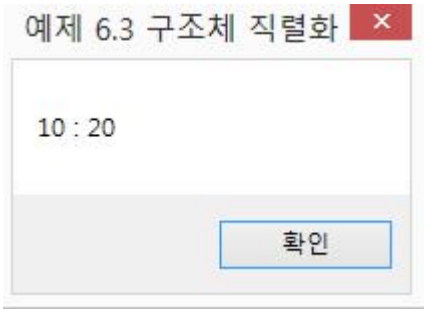
```
// serialize 가 되는 class 만들기-------------------------
struct People : public CObject                              // 1
{
        int a;   int b;
        People() {}  // default 생성자 필요.          // 2
        void Serialize( CArchive& ar )                      // 3
        {
                CObject::Serialize(ar);
                if( ar.IsStoring() )
                        ar << a << b; // 멤버를 넣는다.
                else
```

```
                {
                        ar >> a >> b;
                }
        }
        DECLARE_SERIAL( People )                                //4
};

IMPLEMENT_SERIAL( People, CObject ,1 )              //5
//-------------------------------------------------------------
```

```
void C예제63구조체직렬화View::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
        //1) 파일 생성
        CFile file(TEXT("a.txt"), CFile::modeCreate | CFile::modeWrite);

        //2) file과 연결된 Archive 만들기.
        CArchive ar( &file, CArchive::store);

        int a = 10; RECT r = {10,10,20,20}; CString s = TEXT("hello");

        People p;
        p.a             = 10;
        p.b             = 20;
        ar << &p;                       // 이제 가능...!!!!!!

//      ar << a << r << s; // serialize

        ar.Close();
        file.Close();
        CView::OnLButtonDown(nFlags, point);
```

```
}



void C예제63구조체직렬화View::OnRButtonDown(UINT nFlags, CPoint point)
{
	// TODO: 여기에 메시지 처리기 코드를 추가 및/또는 기본값을 호출합니다.
		CFile file(TEXT("a.txt"), CFile::modeRead);

	// file과 연결된 Archive 만들기.
	CArchive ar( &file, CArchive::load);

	int a ; RECT r; CString s;

	People *p = new People;
//	ar >> a >> r >> s; // serialize
	ar >> p;

	ar.Close();
	file.Close();
	int n    = p->a;
	int n1   = p->b;
	TCHAR temp[20];
	wsprintf(temp, TEXT("%d : %d"), n, n1);
	MessageBox( temp );

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

예제 6.3 구조체 직렬화



7. BITMAP 출력

7.1 Bitmap

비트 단위로 표현된 이미지를 비트맵이라 한다. 비트맵은 특정 이미지를 형성하기 위해 비트들의 배열로 구성되며 크게 두 가지 종류가 있다.

7.1.1 장치 종속 비트맵(Device Dependent Bitmap : DDB)

GDI 오브젝트에서 말하는 비트맵 오브젝트(GDI 오브젝트)는 장치 종속 비트맵(이하 DDB)을 의미하며, DDB는 윈도우즈 3.0 이전부터 사용하던 형태로 16컬러로 색상이 제한된다.

장치에 종속적이므로 출력장치에 따라 비트들의 배열이 달라진다. 즉, 자신의 시스템에서는 잘 나오던 그림이 다른 시스템에서는 변형되어 이상하게 출력될 수 있다.

MFC는 GDI 비트맵(DDB)을 쉽게 다룰 수 있는 클래스 CBitmap 를 제공한다. DDB 별도의 색상정보가 포함되지 않으며, 아래와 같은 BITMAP 구조체로 정의된다.

```
typedef struct tagBITMAP {
    int bmType;              // 비트맵 유형
    int bmWidth;             // 비트맵 폭
    int bmHeight;            // 비트맵 높이
    int bmWidthBytes;        // 래스터 라인
    BYTE bmPlanes;           // 비트맵의 컬러 플레인 수
    BYTE bmBitsPixel;        // 픽셀당 필요한 비트수
    LPVOID bmBits;           // 비트 배열에 대한 포인터
}BITMAP;
```

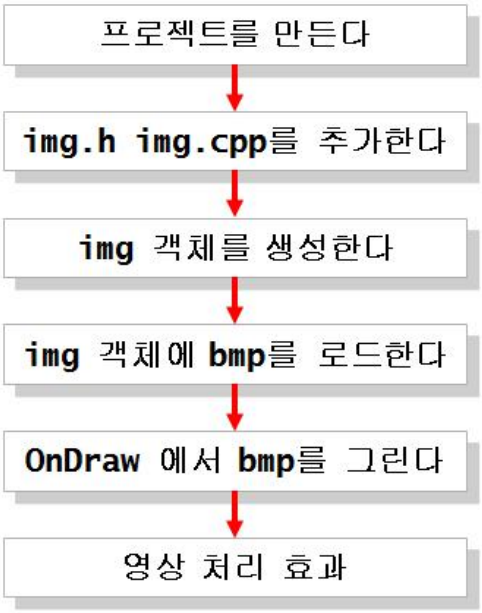
7.1.2 장치 독립 비트맵(Device Independent Bitmap : DIB)

윈도우즈 3.0 부터 지원되는 장치 독립 비트맵(이하 DIB) 은 24비트 트루 컬러를 표현할 수 있다. 장치 독립적이기 때문에 하드웨어 종류가 달라져도 변형없이 출력이 가능하다. DIB는 이미지에 대한 색상 정보나 해상도, 데이터 압축 등을 포함하고 있어 비트맵 데이터를 파일로 저장할 때 유용하다.

일반적으로 BMP 파일로 디스크에 저장하거나 이미지 전송 등을 위해 DIB를 사용한다. 윈도우즈에서는 JPEG, GIF, TIFF, PCX 등의 다른 이미지 포맷도 사용할 수 있지만, Win32 API 가 직접 지원하는 포맷은 DIB 밖에 없다.

7.2 Bitmap 사용하기

아래의 흐름을 통해 Bitmap을 화면에 출력하고자 한다.



7.2.1 비트맵을 화면 중앙에 출력하기

//View 클래스에 맴버 함수 추가하기


```
void C예제71비트맵출력하기View::CenterImage(CDC* pDC)
{
```

```
        CBitmap myBit, *pOldBit;

        CDC memDC;
        BITMAP bm;

        myBit.LoadBitmap(IDB_BITMAP1);
        myBit.GetObject(sizeof(BITMAP), &bm);
        memDC.CreateCompatibleDC(pDC);
        pOldBit = memDC.SelectObject(&myBit);

        CRect rc;
        GetClientRect(rc);
        int x = (rc.Width() - bm.bmWidth)/2;
        int y = (rc.Height() -bm.bmHeight)/2;

        pDC->BitBlt(x,y, bm.bmWidth, bm.bmHeight, &memDC, 0, 0, SRCCOPY);
        memDC.SelectObject(pOldBit);
}
```

```
void C예제71비트맵출력하기View::OnDraw(CDC* pDC)
{
        C예제71비트맵출력하기Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
                return;

        // TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
        CenterImage(pDC);
}
```

예제 7.1 비트맵 출력하기

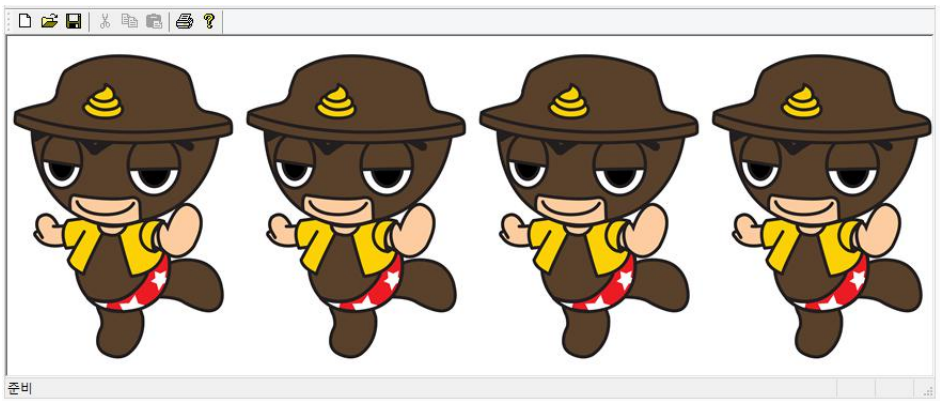
7.2.2 비트맵을 바둑판 모양으로 출력하기

```
void C예제71비트맵출력하기View::TileImage(CDC* pDC)
{
	CBitmap myBit, *pOldBit;

	CDC memDC;
	BITMAP bm;

	myBit.LoadBitmap(IDB_BITMAP1);
	myBit.GetObject(sizeof(BITMAP), &bm);
	memDC.CreateCompatibleDC(pDC);
	pOldBit = memDC.SelectObject(&myBit);

	CRect rc;
	GetClientRect(rc);
```

```
        for(int y=0; y < rc.bottom; y+= bm.bmHeight)
        {
                for(int x =0; x<rc.right; x+= bm.bmWidth)
                {
                        pDC->BitBlt(x,y, bm.bmWidth, bm.bmHeight, &memDC, 0,
0, SRCCOPY);
                }
        }
        memDC.SelectObject(pOldBit);
}
```

```
void C예제71비트맵출력하기View::OnDraw(CDC* pDC)
{
        C예제71비트맵출력하기Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
                return;

        // TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
        TileImage(pDC);
}
```

예제 7.2 비트맵을 바둑판모양으로 출력하기



7.2.3 비트맵을 축소하여 출력하기

```
void C예제71비트맵출력하기View::StretchImage(CDC* pDC)
{
	CBitmap myBit, *pOldBit;

	CDC memDC;
	BITMAP bm;

	myBit.LoadBitmap(IDB_BITMAP1);
	myBit.GetObject(sizeof(BITMAP), &bm);
	memDC.CreateCompatibleDC(pDC);
	pOldBit = memDC.SelectObject(&myBit);
	CRect rc;
	GetClientRect(rc);
	int x = rc.Width();
	int y = rc.Height();
	pDC->StretchBlt(0,0, x, y, &memDC,
		0, 0, bm.bmWidth, bm.bmHeight, SRCCOPY);

	memDC.SelectObject(pOldBit);
}
```

```
void C예제71비트맵출력하기View::OnDraw(CDC* pDC)
{
	C예제71비트맵출력하기Doc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;
	// TODO: 여기에 원시 데이터에 대한 그리기 코드를 추가합니다.
```

```
        StretchImage(pDC);
}
```

예제 7.3 비트맵을 확대 하여 출력하기