Chapter 6.

MFC Architecture

1. CWinApp : 응용 프로그램 클래스

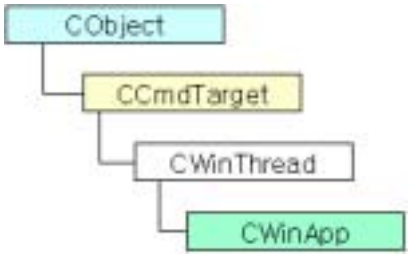


그림. **CWinApp** 클래스

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

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

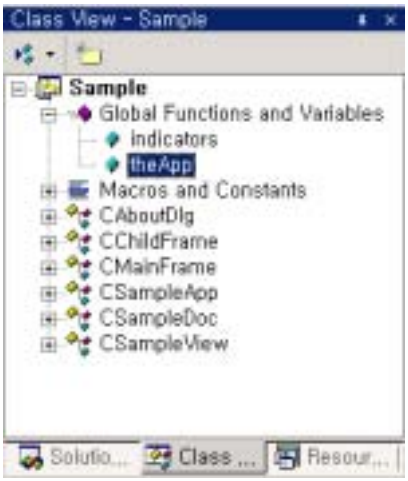


그림. 전역 응용 프로그램 개체

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

다른 윈도우 기반 응용 프로그램과 마찬가지로 MFC로 작성한 응용 프로그램도 **WinMain** 함수를 가집니다. 하지만 이 함수를 직접 작성할 필요는 없습니다. **WinMain** 함수는 MFC에서 제공해주며 응용 프로그램이 시작될 때 호출됩니다. **WinMain** 함수는 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 있습니다.

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

그림. 응용 프로그램 개체의 실행 순서

CWinApp 클래스의 멤버 함수와 별도로 MFC는 **CWinApp** 응용 프로그램 개체를 억세스할 수 있는 전역 함수들을 제공하고 있습니다.

- AfxGetApp : **CWinApp** 개체에 대한 포인터를 반환합니다.
- AfxGetInstanceHandle : 현재 응용 프로그램 개체에 대한 핸들을 반환합니다.
- AfxGetResourceHandle : 응용 프로그램의 리소스에 대한 핸들을 반환합니다.
- AfxGetAppName : 응용 프로그램의 이름을 포함하고 있는 문자열에 대한 포인터를 반환합니다. 만약 **CWinApp** 개체에 대한 포인터를 가지고 있다면 *m_pszExeName* 멤버 변수를 통해 응용 프로그램의 이름을 얻어올 수 있습니다.

1-1. 주요 멤버 함수

CWinApp 클래스에는 몇 가지 오버라이드할 수 있는 중요한 멤버 함수들이 있습니다. 실제로 이들 멤버 함수는 **CWinApp** 클래스가 **CWinThread** 클래스의 멤버 함수를 오버라이드한 것입니다.

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

1-1-1. InitInstance 멤버 함수

응용 프로그램은 윈도우즈 시스템에서 한 번 이상 실행될 수 있습니다. **WinMain** 함수는 새로운 응용 프로그램의 인스턴스가 실행될 때마다 **InitInstance** 함수를 호출합니다.

```
virtual BOOL InitInstance( );
```

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

마법사가 생성한 **InitInstance** 함수는 다음 내용들을 포함하고 있습니다.

○ 다큐먼트, 뷰, 프레임 윈도우를 연결하는 다큐먼트 템플릿을 생성합니다.

○ .INI 파일이나 레지스트리에서 응용 프로그램에 관한 정보를 로드합니다.
○ 하나 이상의 다큐먼트 템플릿을 등록합니다.
○ MDI 형식의 응용 프로그램의 경우 메인 프레임 윈도우를 생성합니다.
○ 명령행 인자를 처리하여 명령행에 지정된 파일을 열거나 빈 다큐먼트를 생성합니다.

다음은 MDI 형식의 응용 프로그램에서 마법사가 생성한 **InitInstance** 함수입니다.

```
BOOL CSampleApp::InitInstance()
{
	// 응용 프로그램 매니페스트가 ComCtl32.dll 버전 6 이상을 사용하여
	// 비주얼 스타일을 사용하도록 지정하는 경우,
	// Windows XP 상에서 반드시 InitCommonControls()가 필요합니다.
	// InitCommonControls()를 사용하지 않으면 창을 만들 수 없습니다.
	InitCommonControls();

	CWinApp::InitInstance();

	// OLE 라이브러리를 초기화합니다.
	if (!AfxOleInit())
	{
		AfxMessageBox(IDP_OLE_INIT_FAILED);
		return FALSE;
	}
	AfxEnableControlContainer();

	// 표준 초기화
	// 이들 기능을 사용하지 않고 최종 실행 파일의 크기를 줄이려면
	// 아래에서 필요 없는 특정 초기화 루틴을 제거해야 합니다.
	// 해당 설정이 저장된 레지스트리 키를 변경하십시오.
	// TODO: 이 문자열을 회사 또는 조직의 이름과 같은
	// 적절한 내용으로 수정해야 합니다.
	SetRegistryKey(_T("로컬 응용 프로그램 마법사에서 생성된 응용 프로그램"));
	// MRU를 포함하여 표준 INI 파일 옵션을 로드합니다.
	LoadStdProfileSettings(4);

	// 응용 프로그램의 문서 템플릿을 등록합니다. 문서 템플릿은
	// 문서, 프레임 창 및 뷰 사이의 연결 역할을 합니다.
	CMultiDocTemplate* pDocTemplate;

	pDocTemplate = new CMultiDocTemplate(IDR_SampleTYPE,
		RUNTIME_CLASS(CSampleDoc),
		RUNTIME_CLASS(CChildFrame), // 사용자 지정 MDI 자식 프레임
```

```
            RUNTIME_CLASS(CSampleView));
        AddDocTemplate(pDocTemplate);

        // 주 MDI 프레임 창을 만듭니다.
        CMainFrame* pMainFrame = new CMainFrame;
        if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
            return FALSE;
        m_pMainWnd = pMainFrame;

        // 접미사가 있을 경우에만 DragAcceptFiles를 호출합니다.
        // MDI 응용 프로그램에서는 m_pMainWnd를 설정한 후
        // 바로 이러한 호출이 발생해야 합니다.
        // 표준 셸 명령, DDE, 파일 열기에 대한 명령줄을 구문 분석합니다.
        CCommandLineInfo cmdInfo;
        ParseCommandLine(cmdInfo);

        // 명령줄에 지정된 명령을 디스패치합니다.
        // 응용 프로그램이 /RegServer, /Register, /Unregserver 또는 /Unregister로
        // 시작된 경우 FALSE를 반환합니다.
        if (!ProcessShellCommand(cmdInfo))
            return FALSE;

        // 주 창이 초기화되었으므로 이를 표시하고 업데이트합니다.
        pMainFrame->ShowWindow(m_nCmdShow);
        pMainFrame->UpdateWindow();
        return TRUE;
}
```

1-1-2. Run 멤버 함수

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

```
virtual int Run( );
```

반환값 : **WinMain** 함수가 반환한 정수값을 반환합니다.

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 처리합니다. 처리할 메시지도 없고 유휴 작업도 없는 경우에는 응용 프로그램은

메시지가 발생될 때까지 응용 프로그램은 수면 상태에 있게 됩니다.

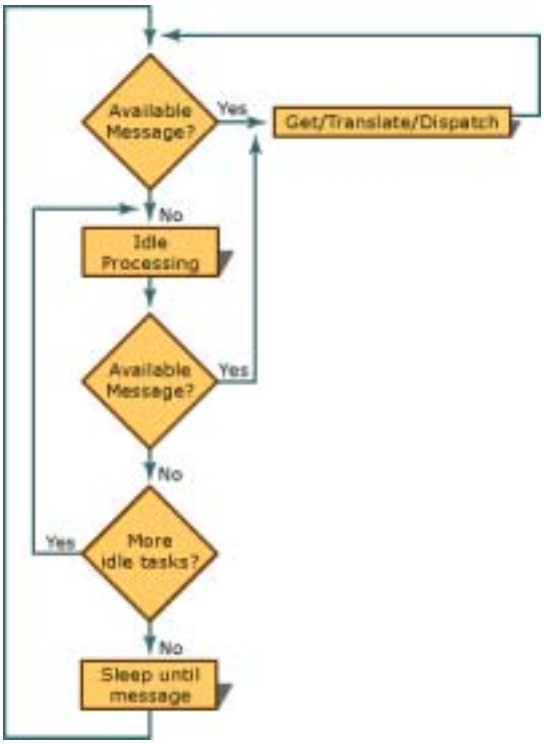


그림. 메시지 루프

읽어온 메시지는 **PreTranslateMessage** 멤버 함수를 통해 필요한 처리를 한 후 표준 키보드 메시지 처리를 위해 **TranslateMessage** 함수로 전달됩니다. 마지막으로 **DispatchMessage** 함수가 호출됩니다.

Run 함수를 오버라이드하여 특별한 처리를 추가할 수 있지만, 오버라이드되는 경우는 거의 없습니다.

1-1-3. ExitInstance 멤버 함수

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

```
virtual int ExitInstance( );
```

반환값 : 응용 프로그램의 종료 코드로 0은 오류가 없는 것을 나타냅니다. 종료 코드는 **WinMain** 함수의 반환값으로 사용됩니다.

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

다음은 **ExitInstance** 함수를 오버라이드한 예입니다.

```
int CMySampleApp::ExitInstance()
{
	// TODO: Add your specialized code here and/or
	// call the base class.

	if ( m_pMySampleMem ) delete m_pMySampleMem;

	DoCleanup();

	return CWinApp::ExitInstance();
}
```

1-4. OnIdle 멤버 함수

유휴 시간 작업을 수행하기 위해서 **OnIdle** 함수를 오버라이드할 수 있습니다.

```
virtual BOOL OnIdle( LONG lCount );
```

Parameter	Description
lCount	응용 프로그램의 메시지 큐가 비어있을 때마다 **OnIdle** 함수가 호출되고 이 때마다 증가되는 카운터를 나타냅니다. 이 카운터는 새로운 메시지가 처리될 때 0으로 초기화됩니다. *lCount* 파라미터를 이용하면 메시지 처리 없이 유휴 상태에 있는 상대적인 시간의 길이를 알 수 있습니다.

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

OnIdle 함수는 응용 프로그램의 메시지 큐가 비어있는 경우 디폴트 메시지 루프에서 호출됩니다. 이 함수를 오버라이드하면 배경 작업을 수행할 수 있습니다.

OnIdle 함수는 더 이상의 유휴 작업이 필요하지 않은 경우 0을 반환합니다. *lCount* 파라미터는 **OnIdle** 함수가 호출될 때마다 증가하며 새로운 메시지가 처리될 때 0으로 초기화됩니다. 여러 가지 배경 작업이 필요한 경우에 이 카운터를 이용하면 여러 유휴 작업을 차례로 수행할 수 있습니다.

유휴 작업의 수행 과정은 다음과 같습니다.

1. 메시지 큐에 메시지가 없으면 MFC는 *lCount* 파라미터를 0으로 설정하여 **OnIdle** 함수를 호출합니다.
2. **OnIdle** 함수는 유휴 작업의 일부를 처리하고 추가적인 유휴 작업이 필요함을 알리기 위해 0이 아닌 값을 반환합니다.
3. 메시지 루프가 메시지 큐를 다시 검사합니다. 여전히 메시지가 없다면 *lCount* 파라미터를 증가시켜서 다시 **OnIdle** 함수를 호출합니다.
4. **OnIdle** 함수가 유휴 작업을 끝내면 0을 반환합니다. 이는 메시지 루프가 메시지 큐에서 다음 메시지를 받을 때까지 **OnIdle** 함수를 호출하지 않도록 합니다. **OnIdle** 함수가 다시 호출될 때에 *lCount* 파라미터는 0으로 초기화됩니다.

사용자의 입력은 **OnIdle** 함수에서 반환하기 전까지 처리되지 않으므로 **OnIdle** 함수에서 오랜 시간이 걸리는 작업을 해서는 안됩니다.

OnIdle 함수는 기본적으로 메뉴 항목이나 툴바 버튼과 같은 사용자 인터페이스 개체를 갱신하고 작업 중에 생성된 임시 개체들을 해제하도록 구현되어 있습니다. 따라서 **OnIdle** 함수를 오버라이드한 경우에는 오버라이드한 함수에서 *lCount*를 이용하여 **CWinApp::OnIdle** 함수를 호출해주어야 합니다. 이 때 기반 클래스의 모든 유휴 작업이 끝날 때까지 즉, 기반 클래스의 **OnIdle** 함수가 0을 반환할 때까지 계속 호출해야 합니다. 기반 클래스의 유휴 작업 처리가 끝나기 전에 수행할 작업이 있다면 카운터에 따라 유휴 시간을 나누어 사용하는 것이 바람직합니다.

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

다음은 *lCount* 카운터를 이용하여 여러 개의 유휴 작업을 서로 다른 우선순위로 관리하는 예입니다.

```
// 여기에서는 4개의 작업이 서로 다른 시간에 수행되고 있습니다.
// Task1 : 메시지 큐에 메시지가 없고, 시스템의 유휴작업이 완료된 경우.
// 항상 실행됩니다. lCount가 0이나 1인 경우는
// Task2 : Task1이 실행된 이후에 실행될 수 있습니다.
// 단, Task1 실행 중에 새로운 메시지가 발생하지 않아야 합니다.
// Task3, Task4 : Task1, Task2가 실행된 이후에 실행될 수 있습니다.
// 역시 Task1 ~ Task3이 실행되는 동안 새로운 메시지가 없어야 합니다.
// Task4는 Task3이 실행되면 항상 실행됩니다.

BOOL CMyApp::OnIdle(LONG lCount)
{
        // 이 예에서는 대부분의 응용 프로그램과 마찬가지로
        // 기반 클래스의 CWinApp::OnIdle 함수가 작업을 완료한 후에
        // 추가적인 유휴 작업을 시행합니다.
        if (CWinApp::OnIdle(lCount)) return TRUE;

        // 기반 클래스의 CWinApp::OnIdle 함수는 lCount가 0이나 1인 경우를
```

```
        // 예약하고 있습니다. 시스템과 이 수준의 시간을 함께 사용하기 위해서는
        // 먼저 위에서 if-문 제거하여 기반 클래스 함수를 직접 호출하고
        // 아래의 switch-문에서 0이나 1인 경우를 추가해주면 됩니다.

        switch (lCount)
        {
        case 2:
                Task1();
                return TRUE; // 다음번에는 Task2가 작업을 수행합니다.
        case 3:
                Task2();
                return TRUE; // 다음번에는 Task3과 Task4가 작업을 수행합니다.
        case 4:
                Task3();
                Task4();
                return FALSE; // 처음부터 다시 시작합니다.
        }
        return FALSE;
}
```

1-2. CWinApp 클래스의 부가 기능

응용 프로그램의 초기화, 메시지 루프의 실행, 종료 등을 관리하는 것 이외에도 응용 프로그램 개체는 몇 가지 부가적인 서비스를 제공합니다.

1-2-1. Shell Registration

기본적으로 MFC 응용 프로그램 마법사는 윈도우 익스플로러나 파일 관리자에서 응용 프로그램이 작성한 데이터 파일을 더블 클릭으로 열 수 있는 기능을 제공합니다. MDI 형식의 응용 프로그램에서 응용 프로그램이 작성하는 데이터 파일의 확장자를 지정하면, 마법사는 **RegisterShellFileTypes** 함수와 **EnableShellOpen** 함수를 **InitInstance** 함수에 추가해 줍니다.

```
// DDE Execute 열기를 활성화합니다.
EnableShellOpen();
RegisterShellFileTypes(TRUE);
```

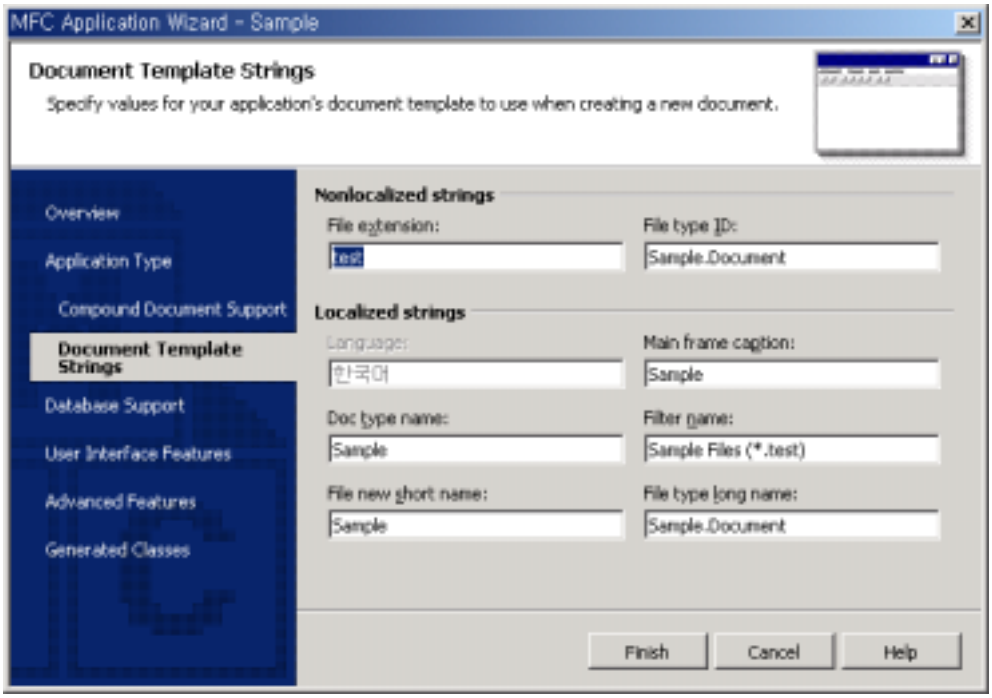


그림. 응용 프로그램의 확장자 등록

RegisterShellFileTypes 함수는 다큐먼트 타입을 시스템이 관리하는 데이터베이스에 등록하고, 확장자와 파일 타입을 연결시켜 줍니다.

EnableShellOpen 함수는 응용 프로그램이 윈도우 익스프로러나 파일 관리자로부터 DDE 명령을 받아서 사용자가 선택한 파일을 열 수 있도록 해줍니다.

1-2-2. 파일 관리자에서 끌어다 놓기

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

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

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

1-2-3. 최근 사용한 파일 목록

사용자가 파일을 열고 닫을 때 응용 프로그램 개체는 최근 작업 파일(Most Recently Used File, MRU File)들의 목록을 유지합니다. 이들 파일 이름은 파일 메뉴에 추가되어 관리됩니다. MFC는 이들 파일 이름을 레지스트리나 .INI 파일에 저장합니다. **InitInstance** 함수에서 호출되고 있는 **LoadStdProfileSettings** 함수는 레지스트리나 .INI 파일에서 응용 프로그램에 관한 정보를 읽어오며,

여기에 최근 작업 파일에 관한 정보도 포함되어 있습니다.

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

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

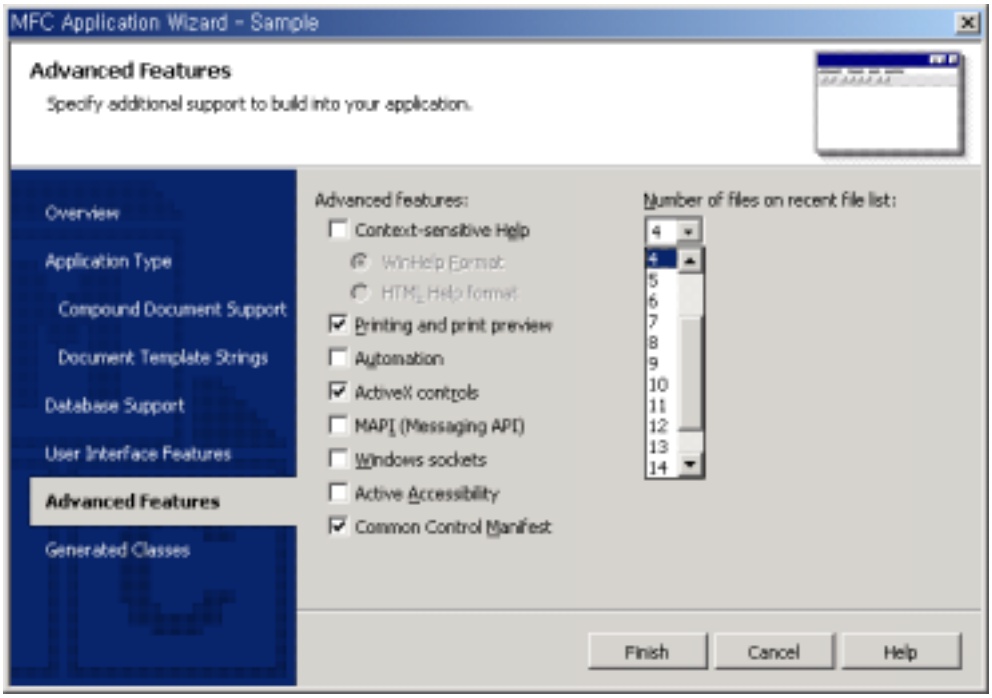


그림. 최근 작업 파일의 수

응용 프로그램의 정보는 윈도우의 버전에 따라 저장되는 방법이 다릅니다.

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

2. 윈도우 개체 (Window Object)

MFC는 윈도우에 대한 핸들인 **HWND**를 캡슐화한 클래스로 **CWnd** 클래스를 제공하고 있습니다. **CWnd** 개체는 C++에서의 윈도우 개체와 동일한 것으로 윈도우를 나타내는 **HWND**를 포함하고 있습니다. **CWnd** 클래스는 윈도우가 가져야하는 기본적인 기능들을 모두 포함하고 있는 클래스로 MFC의 모든 윈도우 개체들은 **CWnd** 클래스를 상속하여 만들어집니다. **CWnd** 유도 클래스들로는 프레임 윈도우, 다이얼로그 박스, 차일드 윈도우, 컨트롤, 컨트롤바, 툴바 등이 있습니다.

MFC는 윈도우의 동작과 윈도우 관리를 위해 몇 가지 기본 구현을 제공하고 있습니다. 제공되는 기본 구현은 **CWnd** 클래스를 상속하여 수정할 수도 있지만, 대부분의 응용 프로그램 클래스들은 **CWnd**

클래스를 직접 상속하지 않고 그 유도 클래스를 상속하여 사용합니다. **CWnd** 클래스는 모든 윈도우에 공통으로 적용되는 특성을 가진 클래스임을 생각해보면 쉽게 이해할 수 있습니다.

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

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

2-1. C++ 윈도우 개체와 HWND의 관계

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

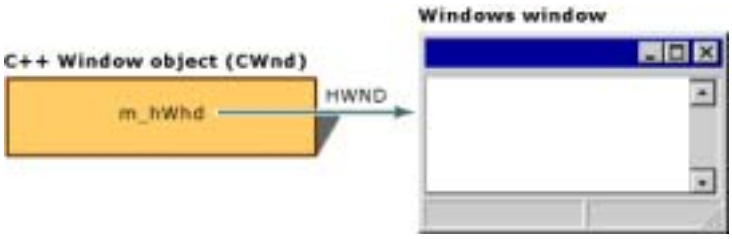


그림. 윈도우 개체와 윈도우

MFC에서 윈도우는 두 단계를 통해 생성되는 것이 일반적입니다. 먼저 윈도우 개체를 생성합니다. 이는 생성자를 통해 다른 C++ 개체와 마찬가지로 생성됩니다. 윈도우 개체가 생성된 후 윈도우가 생성됩니다. 윈도우의 생성은 윈도우 개체의 멤버 함수인 **Create** 함수를 통해 수행됩니다.

2-2. CWnd 유도 클래스

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

2-2-1. 프레임 윈도우 클래스

- CFrameWnd : 단일 다큐먼트와 뷰를 포함하는 SDI 형식의 프레임 윈도우를 위해 사용됩니다. 이 프레임 윈도우는 응용 프로그램을 위한 메인 프레임 윈도우인 동시에 다큐먼트를 위한 프레임 윈도우의 역할을 합니다.
- CMDIFrameWnd : MDI 형식의 메인 프레임 윈도우를 위해 사용됩니다. 메인 프레임 윈도우는 모든 MDI 다큐먼트 윈도우를 포함하며 다큐먼트 윈도우들은 메인 프레임 윈도우의 메뉴를 공유합니다. MDI 프레임 윈도우는 데스크탑에 나타나는 최상위 윈도우입니다.
- CMDIChildWnd : MDI 메인 프레임 윈도우에서 각각의 다큐먼트를 위해 사용됩니다. 각 다큐먼트와 뷰는 차일드 프레임에 연결되어 있습니다. 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 공유합니다.

2-2-2. 기타 CWnd 유도 클래스

프레임 윈도우 이외에도 다른 중요한 클래스들이 **CWnd** 클래스를 상속하고 있습니다.

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

2-3. 윈도우의 생성

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

2-3-1. 윈도우 클래스 등록

"윈도우 클래스"는 윈도우 프로그래밍에서 윈도우를 생성하기 위해 필요한 특성들을 정의합니다. 이는 C++에서의 클래스와 달리 윈도우 생성을 위한 템플릿 또는 모델을 말합니다.

전통적인 윈도우 응용 프로그램에서 윈도우 클래스의 등록은 **RegisterClass** 함수나 **RegisterClassEx** 함수를 통해 이루어집니다. 이들 함수는 윈도우 클래스의 속성을 나타내는 매개변수를 받아서 유일한 윈도우 클래스를 시스템에 등록해 줍니다.

MFC에서는 기존의 방법을 통해서도 윈도우 클래스를 등록할 수 있지만 대부분의 윈도우 클래스 등록이 자동적으로 이루어집니다. MFC는 이미 등록된 여러 윈도우 클래스를 제공하고 있으며 **AfxRegisterWndClass** 함수를 통해 필요한 클래스를 추가로 등록할 수도 있습니다. 등록된 클래스는 **Create** 멤버 함수를 통해 윈도우 생성에 이용됩니다.

```
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 적습니다. 윈도우 등록에 필요한 다른 파라미터들은 디폴트 값을 자동적으로 이용합니다. 함수에서 반환되는 문자열은 MFC가 생성해주는 클래스의 이름으로 파라미터를 기초로 하여 생성됩니다. 따라서 동일한 파라미터를 사용하여 **AfxRegisterWndClass** 함수를 여러 번 호출하면 두 번째 호출 이후로는 처음 등록된 클래스의 이름이 반환됩니다. 반환된 문자열은 다음번 **AfxRegisterWndClass** 함수를 호출할 때까지만 유효한 정적 스트링 버퍼에 대한 포인터이므로 등록한 클래스로 여러 윈도우를 생성할 필요가 있다면 이를 저장해두는 것이 좋습니다.

```
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 됩니다.

DLL에서는 **AfxRegisterClass** 또는 **AfxRegisterWndClass** 함수를 사용하는 것이 중요합니다. Win32는 DLL이 등록한 클래스를 자동적으로 등록 해제하지 않습니다. 따라서 DLL이 종료될 때 명시적으로 등록을 해제해 주어야 합니다. **AfxRegisterClass** 함수는 이러한 등록 해제도 자동적으로 수행해 줍니다. **AfxRegisterClass** 함수는 DLL이 등록한 클래스의 목록을 관리하며 DLL이 종료될 때 자동으로 등록 해제합니다. **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 있습니다.

2-3-2. 윈도우 생성 순서

MFC가 제공하는 모든 윈도우 클래스는 2단계 생성 과정을 거칩니다. 생성 과정에서는 먼저 C++의 **new** 연산자를 사용하여 C++ 개체를 할당하고 초기화합니다. 하지만 이에 해당하는 윈도우는 생성되지 않습니다. 실제 윈도우의 생성은 C++ 개체가 할당된 후 **Create** 멤버 함수를 호출할 때 생성됩니다.

Create 멤버 함수는 윈도우를 생성하고 생성된 윈도우의 **HWND**를 C++ 개체의 *m_hWnd* 멤버 변수에 저장합니다. **Create** 함수를 호출하기 전에 아이콘이나 클래스 스타일 등을 설정하기 위해 **AfxRegisterWndClass** 함수를 사용하여 윈도우 클래스를 등록할 수 있습니다.

프레임 윈도우의 경우에는 **Create** 함수 대신 **LoadFrame** 함수를 사용합니다. **LoadFrame** 함수는 **Create** 함수에 비해 필요한 파라미터의 수가 적습니다. 이는 프레임의 타이틀, 아이콘, 가속키 테이블, 메뉴 등을 리소스에서 얻어오기 때문입니다. 이 때 아이콘, 가속키 테이블, 메뉴는 동일한 아이디를 가져야 하며 디폴트 값은 **IDR_MAINFRAME**입니다.

2-4. 윈도우 개체의 파괴

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

2-4-1. 윈도우 파괴 순서

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

한 가지 주의할 점은 프레임 윈도우나 뷰의 경우 윈도우를 파괴하기 위해 C++의 **delete** 연산자를

사용해서는 안되며 **CWnd** 클래스의 **DestroyWindow** 함수를 사용하여야 한다는 점입니다. 이미 **PostNcDestroy** 함수에서 윈도우 개체를 삭제하고 있으므로 이를 다시 삭제하고자 한다면 오류를 발생시킬 것입니다. 다큐먼트나 응용 프로그램 클래스는 윈도우 클래스가 아니므로 위의 순서에 따르지 않습니다.

2-4-2. CWnd와 HWND의 분리

CWnd 개체에서 윈도우 핸들을 분리하기 위해서 MFC는 **Detach** 함수를 제공하고 있습니다. 이는 윈도우 개체의 소멸자가 윈도우를 파괴하지 못하도록 하기 위해 사용됩니다.

2-5. 윈도우 개체의 사용

CWnd 클래스는 윈도우의 기본적인 동작을 모두 구현하고 있으므로 특별한 경우가 아니면 기본 구현만으로도 필요한 작업을 할 수 있습니다. 이에 비해 응용 프로그램에 따라서 모두 다른 기능 즉, 프로그래머가 윈도우 개체를 이용함에 있어서 주로 다루게 되는 것은 다음 두 가지 동작입니다.

- 윈도우 메시지 처리
- 윈도우 그리기

윈도우 메시지를 처리하기 위해서는 C++ 윈도우 클래스에 메시지들을 매핑해 주어야 합니다. 이는 MFC가 제공하는 메시지 맵을 통해 처리됩니다. **CWnd** 클래스는 메시지를 처리할 수 있는 **CCmdTarget** 클래스의 유도 클래스이므로 생성될 때 메시지 맵을 가지고 있습니다.

윈도우의 그리기는 윈도우가 가지고 있는 데이터를 사용자를 위해 표현하는 것으로 이러한 그리기 작업은 뷰 클래스의 **OnDraw** 함수에서 처리됩니다. 만약 어떤 윈도우가 뷰의 차일드 윈도우라면 뷰의 그리기 작업 중 일부는 차일드 윈도우의 **OnDraw** 함수 호출을 통해 차일드 윈도우가 처리하도록 할 수 있습니다.

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

2-6. 장치 컨텍스트

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

CPaintDC 개체는 **BeginPaint** 함수를 호출하고 장치 컨텍스트를 통해 그리기를 한 후 **EndPaint** 함수를 호출하는 과정을 캡슐화하고 있습니다. **CPaintDC** 클래스의 생성자는 **BeginPaint** 함수를 호출하고 소멸자는 **EndPaint** 함수를 호출합니다. 따라서 단순히 개체를 생성하고 그리기 작업을 한 후 개체를 소멸시키는 것으로 그리기 작업은 끝납니다. 대표적으로 **OnDraw** 함수로 전달되는 매개변수가

CPaintDC 개체로 **OnPrepareDC** 함수를 통해 초기화가 끝난 상태이므로 그리기에만 신경을 쓰면 됩니다. 그리기 작업을 끝낸 장치 컨텍스트는 **OnDraw** 함수에서 반환한 후 MFC에 의해 자동으로 소멸됩니다.

CClientDC 개체는 윈도우의 클라이언트 영역만을 나타내는 장치 컨텍스트입니다. **CClientDC** 클래스의 생성자는 **GetDC** 함수를 호출하고 소멸자는 **ReleaseDC** 함수를 호출합니다. **CWindowDC** 개체는 프레임 윈도우를 포함하는 전체 윈도우를 나타내는 장치 컨텍스트입니다.

CMetaFileDC 개체는 메타 파일로의 그리기를 캡슐화한 것입니다. 이는 **CPaintDC** 개체와는 달리 **OnPrepareDC** 함수를 호출해주어야 합니다.

2-7. 그래픽 개체

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

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 있습니다. 또한 MFC 그래픽 개체 클래스를 그에 대응하는 윈도우의 핸들로 캐스팅할 수 있는 연산자를 가지고 있습니다. 캐스팅된 핸들은 개체가 분리될 때까지 유효합니다. 개체의 **Detach** 함수는 개체에서 핸들을 분리하기 위해 사용됩니다.

다음 코드는 **CPen** 개체를 생성하여 윈도우의 핸들로 변환하는 예입니다.

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

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

1. 스택 프레임에 그래픽 개체를 생성하고 생성된 개체를 **CreatePen**과 같은 생성 함수를 통해 초기화 합니다.
2. 개체를 현재의 장치 컨텍스트로 선택합니다. 이 때 이전 그래픽 개체를 저장해 두어야 합니다.
3. 현재 그래픽 개체로 그리기 작업이 끝난 경우 이전 그래픽 개체를 장치 컨텍스트로 다시 선택하여 원상태로 복구합니다.
4. 실행 범위를 벗어나면 자동으로 스택에 할당된 그래픽 개체는 삭제됩니다. 만약 그래픽 개체가 계속해서 필요하다면 이를 할당해 놓고 필요한 경우에 장치 컨텍스트로 선택할 수 있습니다. 이 경우 더 이상 그래픽 개체가 필요하지 않다면 이를 반드시 삭제하여야 합니다.

2-7-1. 개체의 1단계 또는 2단계 생성

펜이나 브러시와 같은 그래픽 개체들을 생성하는 방법은 두 가지가 있습니다.

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

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

다음은 두 가지 생성 방법을 간단하게 나타낸 것입니다.

```
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
}
```

2-7-2. 장치 컨텍스트로 그래픽 개체의 선택

그래픽 개체를 생성한 후에는 장치 컨텍스트로 생성한 그래픽 개체를 선택해 주어야 합니다. 다음은 펜을 생성하여 그리기를 수행하는 간단한 예입니다.

```
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 임시적입니다. 즉 다음번에 CWinApp 클래스의 OnIdle 함수가 호출되는 경우 삭제됩니다. 하지만 SelectObject 함수에서 반환된 개체를 하나의 함수 내에서만 사용하고 사용 도중 메시지 루프로 제어권을 넘기지 않는다면 아무런 문제가 없습니다.

3. 다큐먼트-뷰 구조

기본적으로 MFC의 응용 프로그램 마법사는 다큐먼트와 뷰 클래스를 포함하는 코드를 만들어 냅니다. 다큐먼트-뷰 구조는 데이터 자체를 그 표현 및 사용자의 조작으로부터 분리하고 있습니다.

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

뷰는 데이터를 표시하고 데이터의 선택, 편집 등 사용자와의 상호작용을 관리합니다. 뷰는 표현할 데이터를 다큐먼트로부터 얻어오며 데이터가 변경된 경우 이를 다시 다큐먼트에 알려줍니다.

다큐먼트-뷰 구조를 이용하지 않을 수도 있지만, 대부분의 응용 프로그램에서 이 구조는 적합합니다. 예를 들어 하나의 데이터를 여러 가지 형식으로 표현하고 싶은 경우 하나의 다큐먼트와 여러 개의 뷰를

사용하여 구현할 수 있습니다. 이는 다큐먼트-뷰 구조가 데이터로부터 그 표현을 분리하고 있기 때문에 가능한 것으로 뷰에 다큐먼트가 포함된 경우라면 데이터의 중복이 발생할 수 있어 비효율적입니다.

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

다큐먼트-뷰 구조는 4개의 핵심 클래스로 구성되어 있습니다.

- **다큐먼트 : CDocument** 클래스는 데이터를 저장하거나 프로그램에서 사용하는 데이터를 제어하기 위한 기능을 제공합니다. 다큐먼트는 파일 메뉴에서 파일 열기 또는 저장 항목과 관련된 데이터 단위를 나타냅니다.
- **뷰 : CView** 클래스는 뷰와 관련된 기본 기능을 제공합니다. 뷰는 다큐먼트에 연결되어 다큐먼트와 사용자 사이의 중개 연학을 합니다. 뷰는 다큐먼트를 화면이나 프린터에 표시하고 사용자의 입력을 다큐먼트로 전달합니다.
- **프레임 윈도우 : CFrameWnd** 클래스는 다큐먼트를 나타내는 하나 이상의 뷰에 프레임을 제공합니다.
- **다큐먼트 템플릿 : CDocTemplete** 클래스는 서로 연관되어 있는 다큐먼트, 뷰 그리고 프레임 윈도우 개체들을 생성하고 이들을 관리하는 책임을 집니다.

다음 그림은 다큐먼트와 뷰의 관계를 나타낸 것입니다.

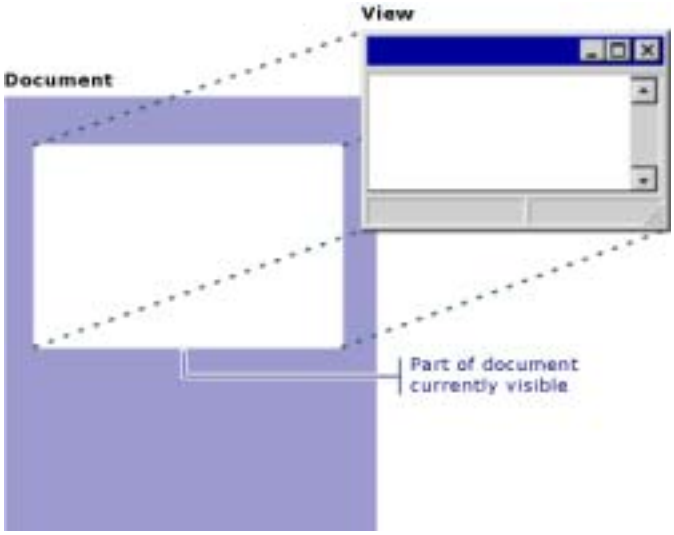


그림. 다큐먼트-뷰 구조

3-1. 다큐먼트-뷰 구조 개요

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

다큐먼트-뷰 쌍에서 뷰는 다큐먼트의 데이터를 억세스하기 위해 여러 가지 방법을 사용할 수 있습니다. 먼저 다큐먼트에 대한 포인터를 반환하는 GetDocument 멤버 함수를 사용할 수 있습니다. GetDocument 함수는 뷰와 연결되어 있는 다큐먼트의 포인터를 반환하므로 다큐먼트의 멤버 변수를

직접 억세스하거나 멤버 함수를 통해 간접 억세스할 수 있습니다. 다른 방법은 뷰 클래스를 다큐먼트 클래스의 **friend** 클래스로 만드는 것입니다.

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

```
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 있습니다. 사용자가 문자열을 표시하고 있는 뷰에 새로운 문자열을 입력한 경우를 생각해 봅시다. 뷰는 다큐먼트에 대한 포인터를 얻고 입력된 문자열을 다큐먼트로 전달하여 다큐먼트가 데이터를 저장하도록 할 것입니다.

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

3-2. 다큐먼트-뷰 구조의 장점

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

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

3-3. 다큐먼트-뷰 구조를 사용하지 않는 경우

MFC는 정보 관리와 데이터 시각적인 표현을 위해 일반적으로 다큐먼트-뷰 구조를 사용합니다. 대부분의 응용 프로그램에서 다큐먼트-뷰 구조는 적절하고 효율적인 방법이라 할 수 있습니다.

다큐먼트-뷰 구조는 데이터를 그 표현과 분리하고 있어서 응용 프로그램 개발을 단순화시키고 불필요한 코드를 줄여줍니다. 하지만 모든 경우에 다큐먼트-뷰 구조가 적합한 것은 아닙니다. 다음 예들을 살펴봅시다.

- C 언어로 작성한 윈도우 프로그램을 변환하는 경우에는 다큐먼트-뷰 구조를 사용하지 않고 변환하기를 원할 수 있습니다. 이런 경우에는 다큐먼트-뷰 구조를 사용하지 않는 것이 변환 과정을 간단하게 할 수 있습니다.
- 간단한 유틸리티와 같이 단순한 작업을 위한 프로그램을 작성하는 경우에는 다큐먼트-뷰 구조 없이도 가능하며, 다큐먼트-뷰 구조를 사용하는 것이 오히려 부담이 될 수 있습니다.
- 원본 코드에서 데이터 자체와 데이터 표현을 위한 코드들이 섞여있는 경우 이를 분리하여 다큐먼트-뷰 구조로 만드는 것은 힘든 작업이 될 수 있습니다. 이런 경우 원본 코드를 그대로 사용할 수 있습니다.

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

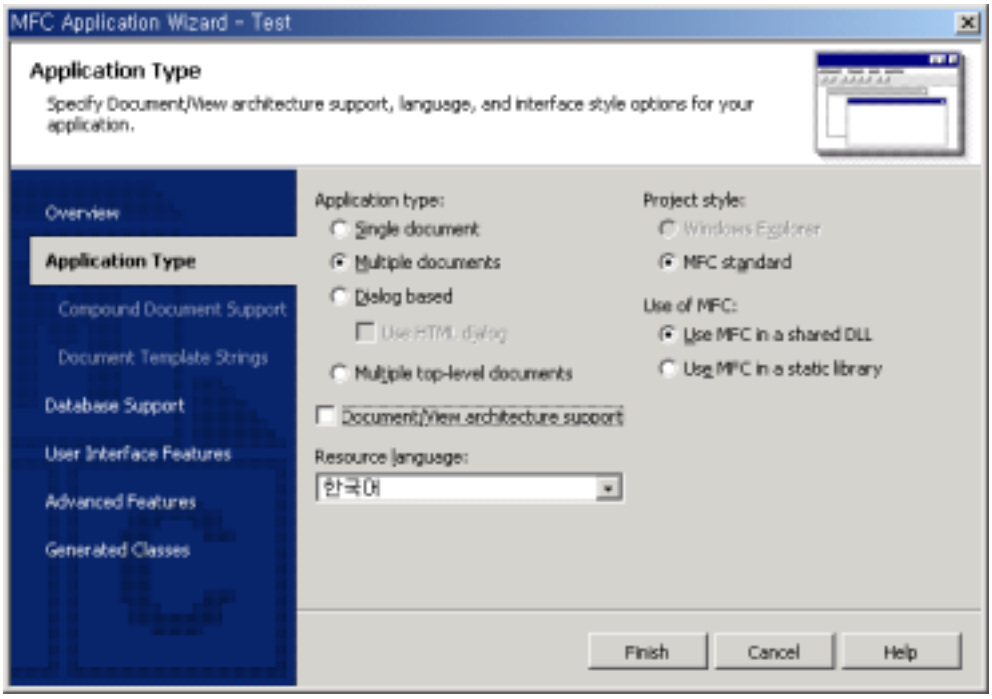


그림. 다큐먼트-뷰 구조 지원 옵션

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

다큐먼트-뷰 구조를 사용하지 않더라도 툴바, 스크롤바, 상태바 등을 사용할 수 있습니다. 하지만 다큐먼트-뷰 구조를 사용하지 않으면 다큐먼트 템플릿을 등록하지 않으며 다큐먼트 클래스도 생성되지 않습니다.

다음 그림은 마법사가 다큐먼트-뷰 구조를 사용하지 않는 SDI 형식의 응용 프로그램을 위해 만든 클래스들입니다. 다큐먼트 클래스는 없지만 다른 클래스들은 다큐먼트-뷰 구조를 사용하는 경우와

동일함을 알 수 있습니다.

그림. 다큐먼트-뷰 구조를 사용하지 않는 SDI 형식 응용 프로그램 클래스

위 그림에서 **CChildView** 클래스는 **CWnd** 클래스에서 유도된 뷰 클래스입니다. MFC는 다큐먼트-뷰 구조를 사용하지 않는 경우에도 뷰 클래스를 생성하여 프레임 윈도우에 위치시키며 데이터를 나타내기 위해 여전히 뷰 클래스를 사용합니다. 뷰 클래스는 프레임 클래스의 멤버 변수로 선언되어 있습니다. 한 가지 눈여겨 볼 점은 뷰 클래스에서의 그리기가 **OnDraw** 함수가 아니라 **OnPaint** 멤버 함수를 통해 구현된다는 점입니다.

다큐먼트-뷰 구조는 응용 프로그램의 기본적인 기능들을 구현해주고 있습니다. 만약 다큐먼트-뷰 구조를 사용하지 않는다면 다큐먼트-뷰 구조에서 지원되는 다음 기능들은 직접 구현하여야 합니다.

- 응용 프로그램의 파일 메뉴에 "새 파일"과 "종료"만이 제공됩니다. "새 파일"의 경우에도 MDI 형식에서만 지원되고 SDI 형식에서는 지원되지 않습니다. 또한 최근 사용한 파일 목록은 지원되지 않습니다.
- 파일 메뉴의 "열기"와 "저장"을 포함하여 응용 프로그램이 제공하는 명령에 대한 핸들러 함수를 직접 구현하여야 합니다. MFC는 이들 명령에 대한 기본 구현을 제공하지만 다큐먼트-뷰 구조를 사용한 경우에 한합니다.
- 제공되는 툴바에는 최소한의 버튼만이 제공됩니다.

3-4. 다큐먼트 클래스의 이용

다큐먼트와 뷰는 함께 사용되어 다음과 같은 일들을 처리합니다.

- 응용 프로그램의 데이터들을 저장, 관리, 표현합니다.
- 데이터 조작을 위한 인터페이스를 제공합니다.
- 파일로 쓰기와 읽기에 관여합니다.
- 인쇄하기와 인쇄 미리보기에 관여합니다.
- 대부분의 명령과 메시지를 처리합니다.

다큐먼트는 특히 데이터 관리와 연관이 있습니다. 데이터는 일반적으로 다큐먼트 클래스의 멤버 변수로 저장되고, 뷰는 이들 변수를 억세스하여 데이터를 표현하고 갱신합니다. 파일로 읽기와 쓰기를 위해서 다큐먼트 클래스는 직렬화(serialization)를 제공하고 있습니다. 이는 **CObject** 클래스에서 제공하는 기능으로 다큐먼트 개체의 현재 상태를 저장하기 위해 사용됩니다. 또한 다큐먼트는 명령 메시지를

처리할 수 있습니다. 하지만 GUI를 갖지 않는 클래스이기 때문에 **WM_COMMAND** 메시지를 제외한 윈도우 메시지는 처리할 수 없습니다. 데이터를 조작하기 위한 마우스나 키보드 메시지들은 뷰에서 처리되는 것이 일반적입니다.

3-4-1. 다큐먼트 클래스의 유도

다큐먼트 클래스는 응용 프로그램의 데이터를 저장하고 있고 이를 관리합니다. 다큐먼트 클래스를 이용하기 위해서는 일반적으로 다음 작업들이 필요합니다.

- **CDocument** 클래스의 유도 클래스를 생성합니다. 이는 마법사가 생성해 줍니다.
- 데이터 저장을 위한 멤버 변수를 추가합니다.
- **CDocument** 클래스의 **Serialize** 멤버 함수를 오버라이드합니다. **Serialize** 함수는 다큐먼트의 데이터를 파일로 읽거나 쓰기 위해 사용됩니다.

이외에도 많이 오버라이드할 수 있는 함수는 많습니다. 특히 **OnNewDocument** 함수와 **OnOpenDocument** 함수는 다큐먼트의 데이터를 초기화하기 위해 오버라이드할 수 있습니다. **DeleteContents** 함수는 동적으로 할당된 데이터를 삭제하기 위해 오버라이드합니다.

3-4-2. 데이터 관리

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

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 있습니다.

뷰는 다큐먼트 개체의 포인터를 통해 다큐먼트의 데이터를 억세스합니다. 다큐먼트와 뷰는 생성될 때 **CDocTemplate** 클래스에 의해 연결되어 있으므로 다큐먼트의 포인터를 얻기 위해서는 **CView** 클래스의 **GetDocument** 멤버 함수를 사용하면 됩니다. 이 때 **GetDocument**가 반환한 포인터는 응용 프로그램의 다큐먼트 클래스로 캐스팅하여 사용하여야 합니다.

잦은 다큐먼트-뷰 클래스 사이의 데이터 교환이나 **public**으로 선언되지 않은 다큐먼트 클래스의 멤버 변수를 억세스하기 위해서는, 뷰 클래스를 다큐먼트 클래스의 **friend**로 선언하여 직접 변수를 억세스할 수도 있습니다.

3-4-3. 데이터의 직렬화 (serialization)

영속성(persistence)에 대한 기본 개념은 개체를 현재의 상태 그대로 저장장치에 저장하였다가 이후 개체를 다시 생성할 때 이 데이터를 읽어 들여 원래의 상태로 복구하는 것을 말합니다. 여기에서 중요한 점은 개체 자신이 자신의 상태를 읽거나 쓰는데 책임이 있다는 점입니다. 따라서 영속적인 클래스의 경우 직렬화를 구현하여야 합니다.

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

MFC의 응용 프로그램 마법사는 **CDocument** 유도 클래스에 **Serialize** 함수를 오버라이드하여 작성해줍니다. 데이터 저장을 위한 멤버 변수들을 정의한 후에 **Serialize** 함수를 구현함으로써 다큐먼트의 데이터들을 파일과 연결된 저장 개체(archive object)로 보낼 수 있습니다. **CArchive** 개체는 표준 C++에서의 입출력 스트림에서 제공하는 **cin**이나 **cout**과 유사하지만 텍스트 형식이 아닌 이진 형식으로 기록하는 점이 다릅니다.

마법사가 생성한 **Serialize** 함수는 파일로 읽기와 쓰기를 구별해 주는 코드만을 생성해줍니다. 실제 데이터의 전송은 다큐먼트의 데이터에 따라 프로그래머가 작성하여야 합니다.

```
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 말합니다. 직렬화는 MFC에서 **CObject** 클래스를 통해 지원되므로 **CObject** 클래스에서 상속된 모든 클래스는 직렬화를 사용할 수 있습니다. **CObject** 클래스는 MFC에서 제공하는 모든 클래스들의 최상위 클래스이므로 MFC의 모든 클래스들은 직렬화를 사용할 수 있습니다.

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

CArchive 개체는 쓰기와 읽기 동작을 위해 삽입(<<)과 추출(>>) 연산자를 오버라이드하고 있습니다. 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

점입니다.

원하는 경우 MFC가 제공하는 직렬화를 사용하지 않을 수도 있습니다. 이 경우에는 직렬화 함수를 호출하는 명령들을 오버라이드하여야 합니다. ID_FILE_OPEN, ID_FILE_SAVE, 그리고 ID_FILE_SAVE_AS 표준 명령이 직렬화를 사용하고 있습니다.

◎ 직렬화에서 다큐먼트의 역할

MFC는 파일 메뉴의 열기, 저장, 다른 이름으로 저장 명령에 대해 자동적으로 다큐먼트의 Serialize 함수를 호출합니다. ID_FILE_OPEN 명령의 경우 응용 프로그램 개체의 핸들러 함수를 호출합니다. 여기에서 사용자는 파일 열기 다이얼로그 박스에서 파일 이름을 선택하고, MFC는 CArchive 개체를 생성하여 다큐먼트 클래스의 Serialize 함수를 호출하게 됩니다. MFC가 이미 파일을 열어 놓았기 때문에 Serialize 함수에서는 데이터를 읽어서 필요한 경우 이를 재구성하기만하면 됩니다.

◎ 직렬화에서 데이터의 역할

일반적으로 클래스 형식의 데이터는 자신을 직렬화할 수 있어야 합니다. 즉 한 개체가 저장 개체로 전달된 경우 전달된 개체는 저장 개체로 자신을 쓰는 방법과 저장 개체로부터 읽는 방법을 알고 있어야 합니다. MFC가 제공하는 데이터 형식의 클래스들은 직렬화 함수를 구현하고 있으므로 MFC가 제공하는 데이터 형식의 멤버 변수들은 CArchive 개체로 전달해주기만 하면 직렬화가 수행됩니다. 새로운 데이터 형식의 클래스를 작성하는 경우에는 직렬화를 구현해주어야 합니다.

◎ MFC의 직렬화 사용하지 않기

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

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

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

Serialize 함수를 완전히 배제하고 싶다면 CWinApp 클래스의 OnOpenDocument 함수와 OnSaveDocument 함수를 오버라이드하면 됩니다.

3-4-4. 다큐먼트에서 명령 메시지의 처리

다큐먼트는 메뉴 항목, 툴바 버튼, 가속키 등에서 발생하는 명령을 처리할 수 있습니다. 기본적으로 CDocument 클래스는 직렬화를 이용하여 "저장"과 "다른 이름으로 저장" 명령을 처리합니다. 이외에도 다큐먼트의 데이터에 영향을 미치는 명령들을 처리할 것입니다. 다큐먼트는 뷰와 마찬가지로 메시지

맵을 가지지만 GUI를 가지지 않으므로 표준 윈도우 메시지는 처리하지 않습니다. 단지 **WM_COMMAND** 메시지만을 처리합니다.

3-5. 뷰 클래스의 이용

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

○ 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 있습니다. 특히 뷰 클래스의 초기화를 위해서 **OnInitialUpdate** 함수를 오버라이드할 수 있으며, 뷰가 화면을 업데이트하기 직전에 필요한 처리를 위해 **OnUpdate** 함수를 오버라이드할 수 있습니다. 여러 페이지의 다큐먼트를 사용한다면 인쇄를 할 때 페이지 수를 설정하기 위해 **OnPreparePrinting** 함수를 오버라이드하면 됩니다.

3-5-1. CView 유도 클래스

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

CView 클래스는 뷰 클래스의 기본 기능을 제공합니다. 뷰는 다큐먼트에 부착되어 다큐먼트와 사용자의 매개 역할을 합니다. 뷰는 다큐먼트의 내용을 화면이나 프린터로 출력하고 사용자의 입력을 다큐먼트로 해석해서 전달합니다.

뷰는 프레임 윈도우의 차일드 윈도우입니다. 분할 윈도우의 경우처럼 하나 이상의 뷰가 프레임 윈도우를 공유할 수 있습니다. 뷰 클래스, 프레임 윈도우 클래스, 다큐먼트 클래스는 **CDocTemplate** 개체에 의해 상호 연결됩니다. 사용자가 새로운 윈도우를 열거나 기존 윈도우를 나누는 경우 MFC는 새로운 뷰를 만들어서 다큐먼트에 추가합니다.

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

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

뷰는 다큐먼트의 데이터를 보여주고 수정하는 책임을 가지고 있지만 데이터를 저장하지는 않습니다. 다큐먼트는 뷰에 필요한 데이터를 제공합니다. 뷰는 다큐먼트의 데이터를 직접 또는 멤버 함수를 통해 억세스할 수 있습니다.

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

CView 클래스를 사용하기 위해서는 먼저 **CView** 클래스의 유도 클래스를 생성하고 **OnDraw** 함수에 화면 출력을 위한 코드를 작성합니다. **OnDraw** 함수는 인쇄와 인쇄 미리보기를 위해서도 사용합니다.

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

CScrollView 클래스 이외에도 MFC는 여러 종류의 **CView** 유도 클래스들을 제공하고 있습니다.

그림. 뷰 클래스 계층도

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

◎ 폼 뷰와 레코드 뷰

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

- CFormView : 레이아웃이 다이얼로그 박스 템플릿으로 정의된 스크롤 뷰입니다. 다이얼로그 박스 템플릿에 기초하여 사용자 인터페이스를 구현하고자 할 때 **CFormView** 클래스를 상속하면 됩니다.
- CRecordView : ODBC 레코드 집합을 나타내는 개체에 대한 연결을 가진 폼 뷰입니다.

- CDaoRecordView : 데이터 억세스 개체(DAO) 레코드 집합을 나타내는 개체에 대한 연결을 가진 폼 뷰입니다.
- COLEDBRecordView : 폼 뷰에 OLE DB에 대한 지원을 추가한 뷰입니다.
- CHtmlView : 웹 브라우징을 지원하는 컨트롤을 포함하는 폼 뷰입니다. 웹 브라우저 컨트롤은 다이나믹 HTML을 지원합니다.
- CHtmlEditView : HTML 문서의 편집 기능을 지원하는 폼뷰입니다.

◎ 컨트롤 뷰

컨트롤 뷰는 특정 컨트롤을 뷰로 표시합니다.

- CCtrlView : 모든 컨트롤 뷰의 베이스 클래스입니다.
- CEditView : 표준 에디트 컨트롤을 뷰로 가집니다.
- CRichEditView : 리치 에디트 컨트롤을 뷰로 가집니다.
- CListView : 리스트 컨트롤을 뷰로 가집니다.
- CTreeView : 트리 컨트롤을 뷰로 가집니다.

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

3-5-2. 뷰에 그리기

응용 프로그램은 거의 모든 데이터 표현 즉, 그리기 작업은 뷰의 **OnDraw** 함수에서 일어납니다. **OnDraw** 함수는 기본 구현이 없으므로 반드시 작성해야만 합니다. **OnDraw** 함수는 다음 작업을 수행합니다.

1. 다큐먼트의 멤버 변수에서 데이터를 얻어옵니다.
2. MFC가 **OnDraw** 함수로 전달하는 장치 컨텍스트 개체를 이용하여 다큐먼트 데이터의 그리기를 수행합니다.

다큐먼트의 데이터가 수정되면 뷰는 변화를 반영하기 위해 다시 그려져야 합니다. 일반적으로 이러한 작업은 뷰를 통해 사용자가 다큐먼트의 데이터를 수정한 경우 일어납니다. 이 때 뷰는 다큐먼트의 **UpdateAllViews** 함수를 호출하여 동일한 다큐먼트와 연결되어 있는 모든 뷰들이 데이터를 업데이트 하도록 알려줍니다. **UpdateAllViews** 함수는 각 뷰의 **OnUpdate** 함수를 호출합니다.

OnUpdate 함수의 기본 구현은 전체 클라이언트 영역을 무효화시킵니다. 다큐먼트에서 수정된 부분에 해당하는 뷰의 클라이언트 영역만을 무효화시키기 위해서는 **OnUpdate** 함수를 오버라이드하면 됩니다.

뷰가 무효화되면 윈도우는 **WM_PAINT** 메시지를 보냅니다. 메시지를 받은 뷰의 **OnPaint** 함수는 **CPaintDC** 클래스의 장치 컨텍스트 개체를 생성하고 뷰의 **OnDraw** 함수를 호출합니다. 일반적으로 **OnPaint** 함수는 오버라이드할 필요가 없습니다.

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

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

```
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 출력합니다. 화면 출력의 경우 전달되는 파라미터 *pDC*는 **CPaintDC** 클래스에 대한 포인터로 생성될 때 **BeginPaint** 함수를 호출하므로 따로 호출할 필요가 없습니다. 또한 장치 컨텍스트가 파괴될 때 **EndPaint** 함수를 자동으로 호출해 줍니다.

3-5-3. 사용자 입력

뷰의 다른 중요한 임무는 사용자의 입력을 받아서 처리하는 것입니다. 뷰는 일반적으로 다음 메시지들을 처리합니다.

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

어느 메시지를 처리할 것인지는 응용 프로그램의 필요에 따라 달라집니다. 응용 프로그램이 뷰 내부에서 마우스의 왼쪽 버튼을 눌러 직선을 그리는 경우를 생각해 보겠습니다. 이를 위해서 뷰는 **WM_LBUTTONDOWN**, **WM_MOUSEMOVE**, 그리고 **WM_LBUTTONUP** 메시지를 처리할 것입니다.

또한 뷰에서는 다큐먼트의 데이터와 관련된 편집 메뉴의 명령들도 처리합니다. 복사, 잘라내기, 붙여넣기 등의 명령은 **CWnd** 클래스의 클립보드 관련 멤버 함수들을 통해 처리됩니다.

3-5-4. 인쇄

뷰는 화면 출력뿐만이 아니라 인쇄도 처리합니다. 이는 윈도우즈 시스템의 장치 컨텍스트가 장치 독립적이기 때문에 가능한 것입니다. 뷰가 인쇄 작업에서 하는 일은 다음과 같은 것들입니다.

○ 뷰 클래스는 화면 출력뿐만이 아니라 인쇄를 위해서도 **OnDraw** 함수를 사용합니다.
○ 여러 페이지로 구성되는 다큐먼트를 인쇄를 위해 페이지 단위로 분할합니다.

3-5-5. 스크롤

MFC는 스크롤되는 뷰와 프레임 윈도우의 크기에 자동으로 맞추어지는 뷰를 지원합니다. **CScrollView** 클래스는 두 종류의 스크롤을 모두 지원하는 **CView** 유도 클래스입니다.

◎ 뷰 스크롤하기
다큐먼트의 크기가 뷰가 한 번에 보여줄 수 있는 크기보다 큰 경우는 허다합니다. 이는 다큐먼트 데이터의 크기가 증가하거나 프레임 윈도우의 크기가 줄어들어서 발생할 수도 있습니다. 이런 경우 뷰는 원하는 다큐먼트의 데이터를 보여주기 위해 스크롤이 필요하게 됩니다.

모든 뷰는 **OnHScroll** 함수와 **OnVScroll** 함수를 통해 스크롤바 메시지를 처리할 수 있습니다. **CView**의 유도 클래스를 사용하는 경우 이들 멤버 함수를 통해 직접 구현함으로써 스크롤 기능을 추가할 수 있지만, **CScrollView**의 유도 클래스를 사용하는 경우에는 추가적인 코드 작성이 없이도 뷰가 제공하는 스크롤 기능을 사용할 수 있습니다.

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 가능합니다. 이 값은 응용 프로그램에 따라 다릅니다. 예를 들어 그래픽 프로그램의 경우 줄의 크기는 픽셀 단위일 수 있지만 문서 편집 프로그램의 경우에는 글자의 높이가 되는 것이 일반적입니다.

◎ 뷰 크기 바꾸기
뷰가 프레임 윈도우의 크기에 맞게 크기가 조절되기를 원하는 경우에도 **CScrollView** 클래스를 사용할 수 있습니다. 논리적인 뷰는 프레임 윈도우의 클라이언트 영역에 맞게 크기가 줄거나 늘어납니다. 크기 조절이 가능한 뷰는 스크롤바를 가지지 않습니다.

3-6. 다중 다큐먼트, 뷰

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

3-6-1. 다중 다큐먼트

MFC 응용 프로그램 마법사는 하나의 다큐먼트 클래스를 생성해 줍니다. 하지만 몇몇 경우에는 하나 이상의 다큐먼트 클래스가 필요할 수 있습니다. 각 다큐먼트는 각각의 다큐먼트 클래스에 의해 표시되며 그와 연결된 서로 다른 뷰를 가질 것입니다. 사용자가 "새 파일" 메뉴 항목을 선택하면,

MFC는 응용 프로그램이 지원하는 다큐먼트의 종류를 다이얼로그 박스를 통해 보여주고 그 중 사용자가 선택한 종류의 다큐먼트를 생성할 것입니다. 각 다큐먼트는 다큐먼트 템플릿 개체에 의해 관리됩니다.

또 다른 다큐먼트 클래스는 마법사를 통해 **CDocument**의 유도 클래스를 생성함으로써 추가할 수 있습니다. 클래스가 생성되면 응용 프로그램 클래스의 **InitInstance** 함수에서 **AddDocTemplate** 함수를 한 번 더 호출하여 추가된 다큐먼트 클래스를 등록해주어야 합니다.

3-6-2. 다중 뷰

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

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

1. 동일한 클래스의 여러 뷰 개체들이 서로 다른 MDI 다큐먼트 프레임 윈도우에 연결되는 경우
   하나의 다큐먼트에 대해 두 번째 프레임 윈도우를 생성하는 방법입니다. 사용자가 "창" 메뉴의 "새 창" 항목을 선택하면 새로운 뷰가 동일한 다큐먼트를 바탕으로 만들어지며, 다큐먼트의 서로 다른 부분을 동시에 살펴보기 위해 사용할 수 있습니다. MFC는 "새 창" 메뉴 항목을 MDI 형식의 응용 프로그램에서 프레임 윈도우와 뷰를 복사하기 위해 사용합니다. 이 경우 차일드 프레임 윈도우의 타이틀은 "*:1", "*:2"와 같이 번호가 붙어서 나타납니다.

그림. 동일한 다큐먼트, 동일한 뷰를 여러 프레임으로 표시

2. 동일한 클래스의 여러 뷰 개체들이 하나의 다큐먼트 프레임 윈도우에 연결되는 경우
   분할 윈도우는 뷰 영역을 여러 영역으로 나누어줍니다. 이는 동일한 뷰 클래스의 여러 개체를 하나의 다큐먼트에 대해 생성하는 것으로 위의 경우와 프레임 윈도우가 하나라는 것을 제외하고는 동일합니다.
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 생성됩니다. 이는 동일한 다큐먼트를 서로 다른 형식으로 나타내기 위해 사용됩니다.

다음 그림은 위에서 설명한 3가지 방법을 나타낸 것입니다.

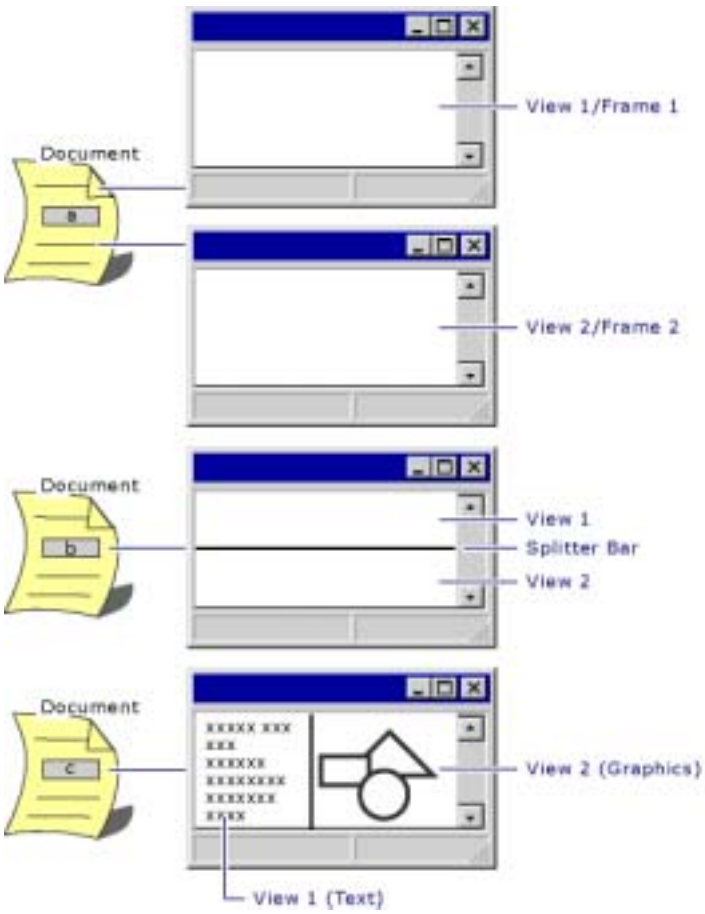


그림. 다중 뷰

3-6-3. 분할 윈도우

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

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

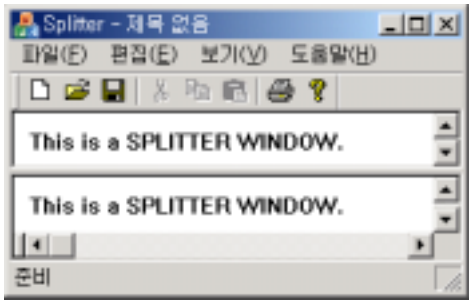


그림. 분할 윈도우

3-7. 다큐먼트와 뷰의 초기화 및 해제

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

- MFC는 다큐먼트와 뷰를 초기화합니다. 하지만 다큐먼트에 추가한 멤버 변수들에 대한 초기화는 프로그래머의 몫입니다.
- MFC는 사용이 끝난 다큐먼트와 뷰를 자동으로 해제합니다. 하지만 다큐먼트와 뷰의 멤버 함수에서 힙에 할당한 메모리를 해제하는 것은 프로그래머의 몫입니다.

전체 응용 프로그램에서 필요한 초기화 및 해제를 위해 적합한 함수는 응용 프로그램 클래스의 **InitInstance**와 **ExitInstance** 함수입니다.

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

1. 동적 생성 과정에서 다큐먼트의 생성자가 호출됩니다.
2. 새로운 다큐먼트에 대해서 **OnNewDocument** 함수나 **OnOpenDocument** 함수가 호출됩니다.
3. 사용자는 다큐먼트와 뷰를 통해 상호 작용합니다. 뷰는 다큐먼트에 데이터의 변화를 알려주고 다큐먼트는 뷰에 업데이트를 알려줍니다.
4. MFC는 **DeleteContents** 함수를 호출하여 다큐먼트와 관련된 데이터를 삭제합니다.
5. 다큐먼트의 소멸자가 호출됩니다.

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

3-7-1. 다큐먼트와 뷰의 초기화

다큐먼트는 2가지 방법을 통해 생성됩니다. 따라서 다큐먼트 클래스는 2가지 방법을 모두 지원해야 합니다. 먼저 사용자는 "새 파일" 메뉴 항목을 통해 빈 다큐먼트를 생성할 수 있습니다. 이 경우 다큐먼트의 초기화는 **OnNewDocument** 함수에서 시행됩니다. 두 번째는 사용자가 "파일 열기" 메뉴 항목을 통해 다큐먼트를 생성하고 파일로부터 내용을 읽어 들입니다. 이 경우 다큐먼트의 초기화는 **OnOpenDocument** 함수에서 시행됩니다. 두 경우의 초기화가 동일하다면 두 함수가 공통으로 호출하는 함수를 작성하여 사용할 수도 있고 **OnOpenDocument** 함수에서 **OnNewDocument** 함수를 호출하여 사용할 수도 있습니다.

뷰는 다큐먼트가 생성된 이후에 생성됩니다. 뷰를 초기화하기에 가장 좋은 시점은 다큐먼트, 프레임 윈도우, 뷰가 모두 생성된 이후입니다. 뷰의 초기화는 **CView** 클래스의 **OnInitialUpdate** 함수를 통해 이루어집니다. 뷰를 다시 초기화하거나 다큐먼트의 변화에 따라 초기화가 바뀌는 경우에는 **OnUpdate** 함수를 사용하면 됩니다.

3-7-2. 다큐먼트와 뷰의 해제

다큐먼트를 닫을 때 MFC가 처음으로 호출하는 함수는 **DeleteContents** 함수입니다. 만약 작업 중에

힙에 할당한 메모리가 있다면 **DeleteContents** 함수는 이를 해제하는 가장 좋은 장소가 됩니다. 한 가지 주의할 점은 다큐먼트의 데이터를 소멸자에서 해제해서는 안된다는 점입니다. SDI의 경우에는 다큐먼트 개체가 재사용되기 때문에 할당한 메모리가 해제되지 않습니다.

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

4. 프레임 윈도우

윈도우 기반의 응용 프로그램을 실행할 때, 사용자는 프레임 윈도우에 표시된 다큐먼트와 상호작용을 하게 됩니다. 다큐먼트 프레임 윈도우는 프레임 자체와 프레임 내의 내용의 2가지 요소로 구성됩니다. 다큐먼트 프레임 윈도우는 단일 다큐먼트 인터페이스(SDI) 프레임 윈도우와 다중 다큐먼트 인터페이스(MDI) 차일드 윈도우가 있습니다.

MFC는 뷰를 관리하기 위해 프레임 윈도우를 사용합니다. 즉, 프레임 자체와 그 내부의 내용은 각기 별개의 MFC 클래스로 구현되고 관리되며, 뷰 윈도우는 프레임 윈도우의 차일드 윈도우입니다. 다큐먼트와 관련된 사용자와의 상호작용은 뷰의 클라이언트 영역에서 일어나며, 프레임 윈도우는 뷰에 프레임을 추가하고, 타이틀바, 메뉴, 최소/최대화 버튼 등을 관리합니다. 다음 그림은 프레임 윈도우와 뷰의 관계를 나타낸 것입니다.

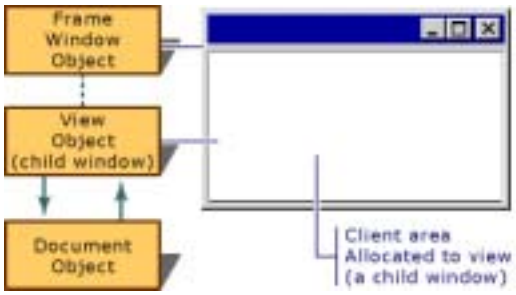


그림. 프레임 윈도우와 뷰

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

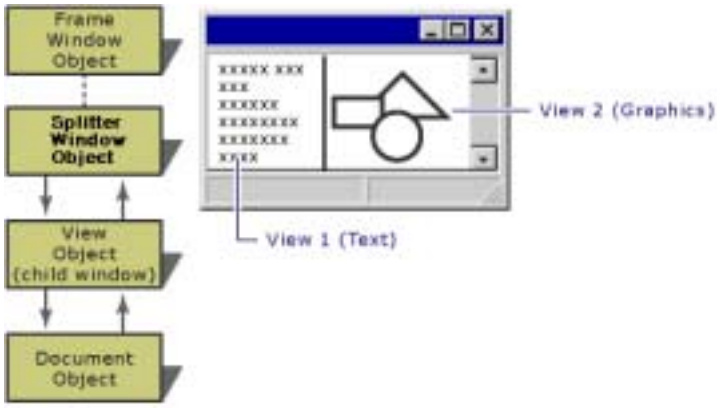


그림. 분할 윈도우

4-1. 프레임 윈도우 클래스

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

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

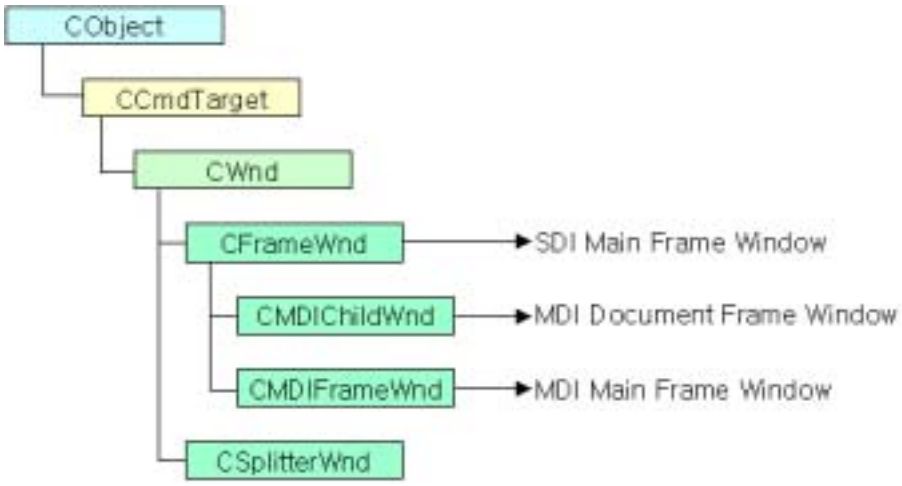


그림. 프레임 윈도우

응용 프로그램 마법사로 프로그램의 골격을 생성하면 마법사는 응용 프로그램을 위한 메인 프레임 윈도우와 MDI 형식의 경우에는 별도로 다큐먼트 프레임 윈도우를 만들어줍니다.

프레임 윈도우는 뷰를 둘러싸고 있는 프레임을 제공하는 것 이외에도 다양한 일을 합니다. 예를 들어 차일드 윈도우를 배치하는 것도 프레임 윈도우의 몫입니다. 차일드 윈도우에 포함되는 차일드 윈도우로는 컨트롤 바, 뷰, 그리고 클라이언트 영역 내부에 있는 윈도우나 컨트롤이 포함됩니다.

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

4-2. 프레임 윈도우 스타일

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

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

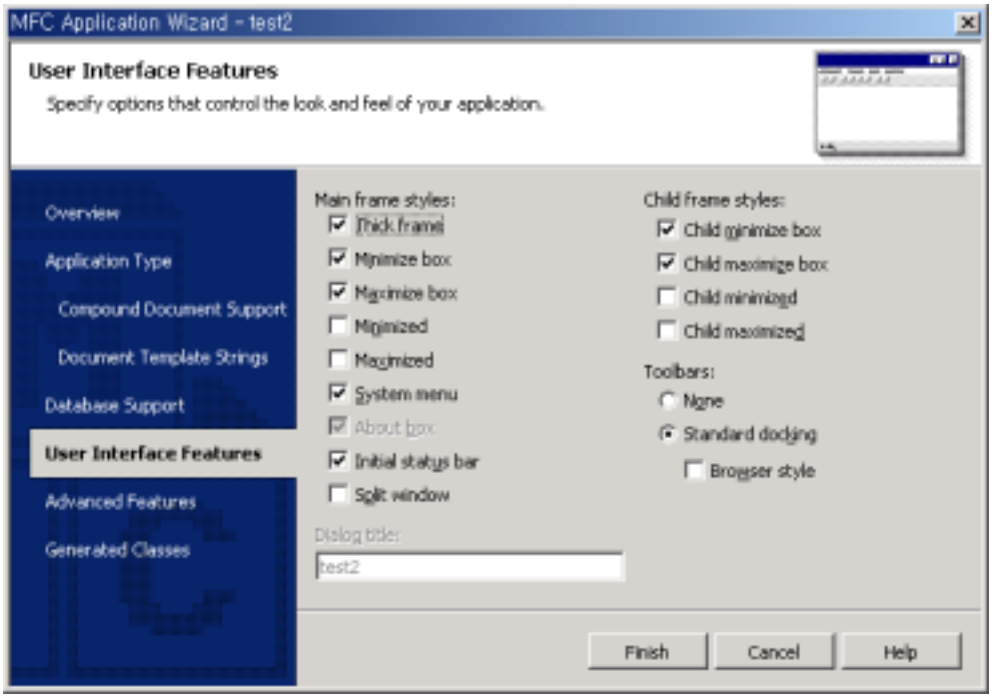


그림. 프레임 윈도우 속성

마법사가 이미 골격 생성을 끝낸 경우에는 **PreCreateWindow** 가상 함수의 오버라이딩을 통해 스타일을 수정할 수 있습니다. **PreCreateWindow** 함수는 윈도우가 생성되기 직전에 호출되어 **CDocTemplate** 클래스가 내부적으로 관리하는 윈도우 생성 작업을 제어할 수 있도록 해줍니다. **PreCreateWindow** 함수의 파라미터인 **CREATESTRUCT** 구조체 값을 수정함으로써 응용 프로그램은 메인 프레임 윈도우의 속성을 변경할 수 있습니다.

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

```
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;
}
```

4-2-1. SDI 프레임 윈도우 스타일의 수정

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

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

```
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);
}
```

4-2-2. MDI 차일드 윈도우 스타일의 수정

MDI 차일드 윈도우의 디폴트 스타일은 **WS_CHILD**, **WS_OVERLAPPEDWINDOW**, 그리고 **FWS_ADDTOTITLE** 스타일의 조합으로 표시됩니다. 차일드 윈도우의 스타일을 수정하기 위해서는 위의 SDI의 경우와 마찬가지로 **PreCreateWindow** 함수를 오버라이드하면 됩니다.

다음은 최대화 버튼이 없는 차일드 윈도우를 생성하는 예입니다.

```
BOOL CMyChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
        // 최대화 버튼이 없는 차일드 윈도우 생성
        cs.style &= ~WS_MAXIMIZEBOX;

        // 기반 클래스의 함수를 호출
        return CMDIChildWnd::PreCreateWindow(cs);
}
```

4-3. 프레임 윈도우의 이용

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

4-3-1. 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 있습니다. 이들이 생성되는 순서는 다음과 같습니다.

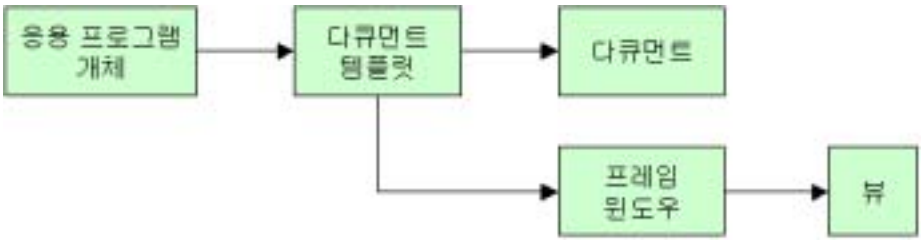


그림. 기본 클래스 개체들의 생성 순서

다큐먼트-뷰 구조의 생성은 **CDocTemplate** 개체가 프레임 윈도우, 다큐먼트, 뷰 개체를 생성하고 연결함으로써 이루어집니다. **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 줍니다. 다큐먼트 템플릿은 이러한 개체들 간의 관계를 저장하였다가 뷰와 다큐먼트를 위해 프레임 윈도우를 생성할 때 이용합니다.

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

사용자가 다큐먼트를 생성하는 명령을 선택하면, MFC는 다큐먼트 템플릿이 다큐먼트 개체, 뷰, 그리고 프레임 윈도우를 생성하도록 합니다. 다큐먼트 프레임 윈도우가 생성할 때 SDI의 경우는 **CFrameWnd** 클래스의 유도 클래스를 MDI의 경우에는 **CMDIChildWnd** 클래스의 유도 클래스를 생성합니다. 이후 MFC는 프레임 윈도우 개체의 **LoadFrame** 멤버 함수를 사용하여 리소스에서 윈도우 생성에 필요한 정보를 얻어옵니다. 윈도우가 생성될 때 윈도우의 핸들은 프레임 윈도우 개체와 연결됩니다. 이후 다큐먼트 프레임 윈도우의 차일드 윈도우로 뷰가 생성됩니다.

한 가지 주의할 점은 프레임 윈도우가 생성된 이후에 뷰가 생성된다는 점입니다. 일반적으로 **CWnd** 유도 클래스의 생성자에서는 차일드 윈도우를 생성할 수 없습니다. 생성자가 호출된 시점에서는 아직 **CWnd** 개체와 연관된 윈도우가 생성되지 않아서 **HWND**가 유효하지 않기 때문입니다. 따라서 차일드 윈도우의 생성과 같은 윈도우와 관련된 초기화는 **OnCreate** 메시지 핸들러에서 해주는 것이 일반적입니다. 이 함수는 윈도우가 생성된 이후 화면에 나타나기 전에 호출됩니다.

4-3-2. 프레임 윈도우의 파괴

MFC는 다큐먼트-뷰 구조와 관련하여 윈도우의 생성뿐만이 아니라 파괴도 관리합니다. 하지만 추가적인 윈도우를 생성한 경우에 그 파괴의 책임은 프로그래머에게 있습니다.

MFC에서 사용자가 프레임 윈도우를 닫으면 **WM_CLOSE** 메시지가 발생하고 **OnClose** 핸들러가 호출됩니다. 이 핸들러의 기본 구현은 **DestroyWindow** 함수를 호출하여 윈도우를 파괴합니다. 윈도우가 파괴될 때 마지막으로 호출되는 함수는 **OnNcDestroy** 함수로 윈도우의 비-클라이언트 영역이 파괴된 후 **WM_NCDESTROY** 메시지의 핸들러로 호출됩니다. 윈도우가 파괴된 이후에 **OnNcDestroy** 함수는 **PostNcDestroy** 함수를 호출합니다. 프레임 윈도우에서 **PostNcDestroy** 함수의 기본 구현은 C++ 윈도우 개체를 삭제합니다. 따라서 프레임 윈도우를 파괴할 때는 C++의 **delete** 연산자를 사용해서는 안되며 반드시 **DestroyWindow** 함수를 사용하여야 합니다.

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

4-3-3. MDI 차일드 윈도우

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

MDI 프레임 윈도우에서 프레임 윈도우는 **MDICLIENT** 윈도우를 관리하고 컨트롤바와 함께 그 위치를

관리하고, **MDICLIENT** 윈도우는 MDI 차일드 프레임 윈도우를 관리합니다. 다음 그림은 이들 윈도우들의 관계를 나타낸 것입니다.

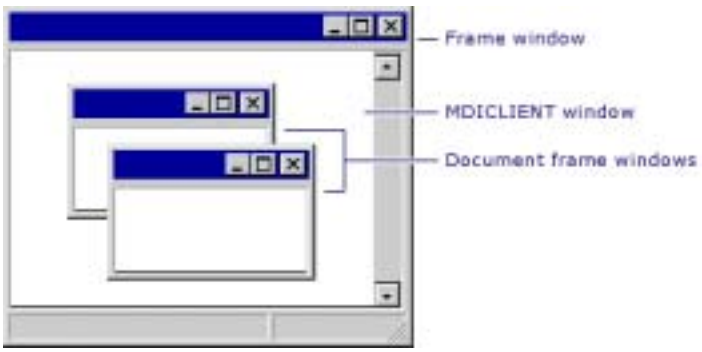


그림. MDI 프레임 윈도우와 차일드 윈도우

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

4-3-4. 뷰 관리

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

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

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

4-3-5. 메뉴, 컨트롤바, 가속키 관리

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

○ 메뉴 관리

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

MDI 응용 프로그램의 경우 MDI 프레임 윈도우는 메뉴바와 캡션을 관리합니다. MDI 프레임 윈도우는 차일드 윈도우가 없을 때 메뉴바로 사용할 디폴트 메뉴를 포함하고 있습니다. 활성 차일드 윈도우가

있는 경우에 MDI 프레임 윈도우는 활성 MDI 차일드 윈도우를 위한 메뉴로 메뉴바를 교체합니다. 만약 MDI 응용 프로그램이 다중 다큐머트 타입을 지원한다면 다큐먼트의 종류에 따라 메뉴와 캡션이 바뀝니다. 동일한 다큐먼트 타입의 여러 MDI 차일드 윈도우는 메뉴를 공유합니다.

다음 그림은 MDI 응용 프로그램에서 차일드 윈도우가 있는 경우와 없는 경우를 비교한 것입니다. 그림을 살펴보면 메뉴와 타이틀바는 물론 툴바의 버튼 상태도 변화한 것을 볼 수 있습니다.

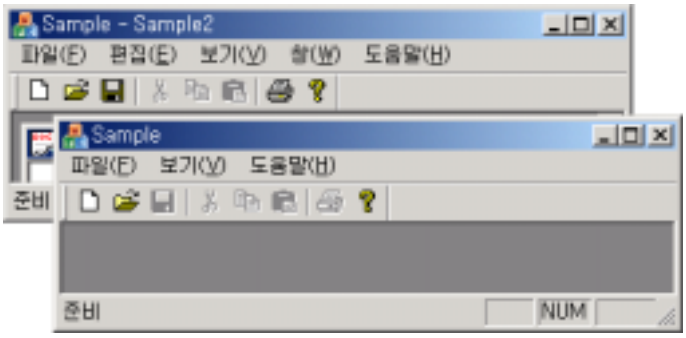


그림. 차일드 윈도우에 따른 메뉴의 변화

CMDIFrameWnd 클래스는 기본적인 명령에 관한 디폴트 구현을 제공하고 있습니다. 특히 '새 파일'에 해당하는 **ID_WINDOW_NEW** 명령은 다큐먼트는 물론 새로운 프레임 윈도우와 뷰를 생성하도록 구현되어 있습니다.

○ 상태바 관리

프레임 윈도우는 클라이언트 영역 내에서 상태바의 위치와 상태바의 지시자(indicator)들을 관리합니다. 프레임 윈도우는 메뉴 항목이나 툴바 버튼을 선택한 경우 상태바의 메시지 영역에 문자열을 출력해줍니다. 출력되는 문자열은 메뉴 편집기에서 메뉴의 'Prompt' 속성을 통해 지정할 수 있습니다.

다음 그림은 파일 열기 메뉴에 지정된 문자열을 보여주고 있습니다. 문자열은 두 부분으로 구성되어 있고 '\n'으로 구분되어 있습니다. 첫 번째 부분은 상태바에 출력되는 내용이고 두 번째 부분은 툴바 버튼에 대한 풍선 도움말로 출력됩니다.

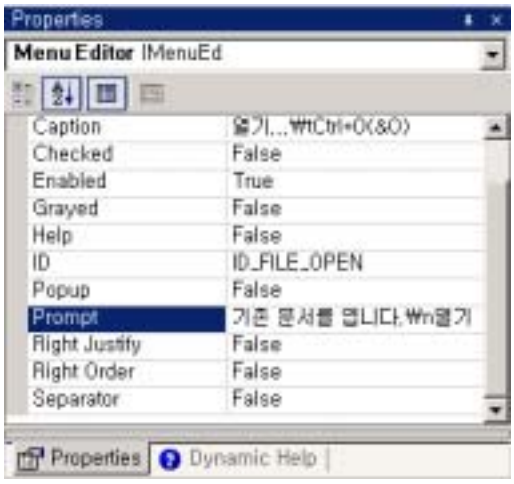


그림. 메뉴의 Prompt 속성

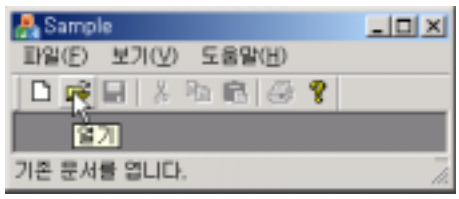


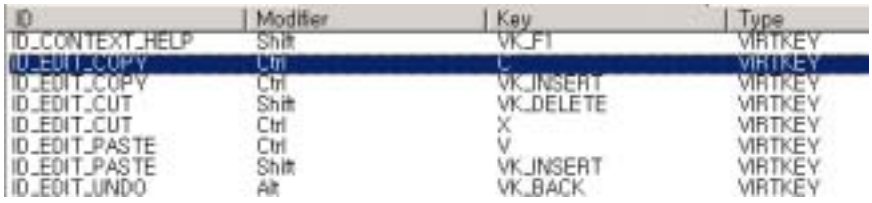
그림. 상태바 메시지와 툴바 버튼의 풍선 도움말

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

○ 가속키 관리

프레임 윈도우는 옵션으로 가속키 테이블을 관리하여 가속키 변환을 자동으로 수행해줄 수 있습니다. 가속키는 메뉴 명령을 발생시킵니다.

리소스의 'Accelerator' 항목에 가속키들이 정의되어 있으며, 메뉴의 'Caption' 항목에 가속키에 해당하는 키가 표시되어 있습니다.



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

그림. 가속키 테이블

그림. 메뉴에서의 가속키 정의

4-3-6. 끌어 놓기 (Drag-and-Drop)

프레임 윈도우는 윈도우 익스플로러나 파일 관리자와의 관계를 관리합니다.

CWinApp 클래스의 InitInstance 함수를 오버라이드하고 메인 프레임 윈도우의 CWnd::DragAcceptFiles 멤버 함수를 호출해줌으로써 프레임 윈도우는 익스플로러나 파일 관리자에서 끌어다 놓은 파일을 열 수 있습니다.

4-4. CFrameWnd 클래스

CFrameWnd 클래스는 단일 다큐먼트 인터페이스(SDI)의 오버랩 또는 팝업 프레임 윈도우를 위한 기본 기능을 제공합니다. 일반적으로 응용 프로그램에서 프레임 윈도우를 생성하기 위해서는 CFrameWnd 클래스의 유도 클래스를 사용합니다.

프레임 윈도우는 두 단계로 생성됩니다. 먼저 생성자를 통해 CFrameWnd 개체가 생성됩니다. 하지만 생성자는 실제 윈도우를 생성하지는 않습니다. 실제 윈도우는 Create 함수나 LoadFrame 함수를 통해 생성되어 CFrameWnd 개체에 연결됩니다. CFrameWnd 개체는 *m_hWnd* 멤버 변수를 통해 윈도우를 억세스할 수 있습니다.

프레임 윈도우를 생성하는 방법은 3가지가 있습니다.

- 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 합니다. 윈도우 개체를 생성하고 Create 함수를 호출하기 전에 아이콘이나 프레임 윈도우의 스타일을 지정하기 위해 AfxRegisterWndClass 함수를 사용하여 윈도우 클래스를 등록할 수 있습니다.

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, 가속키 테이블, 메뉴 등은 리소스에서 얻어와 사용합니다. LoadFrame 함수를 통해 리소스를 얻어오기 위해서는 리소스들이 동일한 아이디를 가져야 합니다. 기본적으로 IDR_MAINFRAME이라는 아이디를 가집니다.

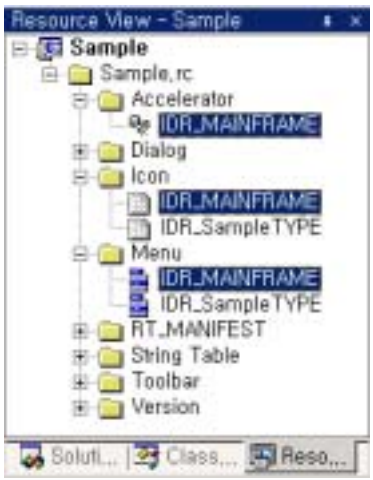


그림. 프레임 윈도우의 리소스

CFrameWnd 개체가 뷰와 다큐먼트를 포함하고 있다면 MFC는 자동적으로 뷰와 다큐먼트를 생성해줍니다. 응용 프로그램 개체가 생성하는 **CDocTemplate** 개체가 프레임 윈도우, 뷰, 다큐먼트 개체를 생성하고 뷰와 다큐먼트를 연결하는 역할을 합니다. **CDocTemplate**의 생성자에는 다큐먼트, 프레임, 뷰에 대한 파라미터를 **CRuntimeClass** 형식으로 지정합니다. **CRuntimeClass** 개체는 새로운 프레임을 동적으로 생성하기 위해 MFC에서 사용합니다.

```
// 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 합니다.

CFrameWnd 클래스의 디폴트 구현에는 다음 내용들이 포함되어 있습니다.

- 프레임 윈도우는 윈도우의 현재 활성 상태나 입력 포커스와 무관하게 현재 활성 뷰를 관리합니다. 프레임 윈도우가 활성화되면 활성 뷰는 **CView::OnActivateView** 함수에 의해 자동으로 활성화됩니다.
- 명령 메시지와 **CView** 클래스의 **OnSetFocus**, **OnHScroll**, **OnVScroll** 등의 함수에서 처리되는 많은 프레임 통지 메시지들은 프레임 윈도우가 현재 활성 뷰로 전달해 줍니다.
- 현재의 활성 뷰나 MDI의 경우에는 현재의 활성 차일드 프레임 윈도우가 프레임 윈도우의 캡션을 설정할 수 있습니다. 이 기능은 프레임 윈도우의 **FWS_ADDTOTITLE** 스타일의 설정 여부에 따릅니다.
- 프레임 윈도우는 프레임 윈도우의 클라이언트 영역 내에 존재하는 컨트롤 바, 뷰, 그리고 다른 차일드 윈도우들의 위치를 관리합니다. 또한 프레임 윈도우는 유휴 시간에 툴바와 컨트롤 바의 버튼을 업데이트하며, 툴바와 상태바를 보이거나 보이지 않게 하는 디폴트 구현을 제공합니다.
- 프레임 윈도우는 메인 메뉴 바를 관리합니다. 또한 팝업 메뉴가 보여지는 경우 프레임 윈도우는 **UPDATE_COMMAND_UI** 메카니즘을 통해 메뉴의 인에이블, 디스에이블, 체크 등의 상태를

결정합니다. 사용자가 메뉴 항목을 선택하면 프레임 윈도우는 상태바에 선택된 항목에 대한 메시지를 보여줍니다.

- 프레임 윈도우는 가속키 테이블을 옵션으로 가질 수 있습니다.
- 프레임 윈도우는 문맥 감지 도움말에 사용되는 도움말 아이디를 옵션으로 가질 수 있습니다. 프레임 윈도우는 문맥 감지 도움말(SHIFT+F1)이나 인쇄 미리보기 모드 등 세미모달 상태를 제어할 책임을 집니다.
- 프레임 윈도우는 파일 관리자에서 끌어다 놓은 파일을 열 수 있습니다. 또한 응용 프로그램에 파일 확장자가 등록된 경우 프레임 윈도우는 사용자의 더블 클릭이나 **ShellExecute** 함수에 의해 발생하는 DDE의 열기 명령을 처리합니다.
- 프레임 윈도우가 응용 프로그램의 메인 윈도우(**CWinThread::m_pMainWnd**)인 경우, 사용자가 응용 프로그램을 종료하면 프레임 윈도우는 수정된 다큐먼트를 저장할 것인지를 물어봅니다.
- 프레임 윈도우가 응용 프로그램의 메인 윈도우인 경우, 프레임 윈도우는 WinHelp를 실행하는 컨텍스트가 됩니다. 프레임 윈도우를 닫으면 실행 중인 WinHelp도 종료됩니다.

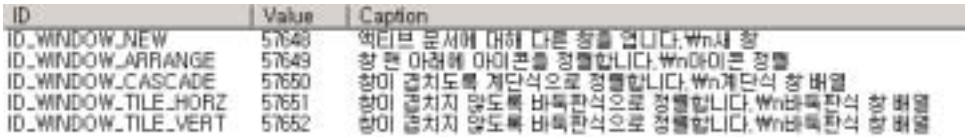
4-5. CMDIFrameWnd 클래스

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

MDIFrameWnd 클래스는 **CFrameWnd** 클래스의 유도 클래스이지만 **CMDIFrameWnd** 유도 클래스들은 동적으로 생성하지 않기 때문에 **DECLARE_DYNCREATE** 매크로를 선언하지 않아도 됩니다.

CMDIFrameWnd 클래스는 기본 구현을 **CFrameWnd** 클래스로부터 상속하고 있습니다. 여기에 다음 특성들을 추가적으로 가집니다.

- MDI 프레임 윈도우는 **MDICLIENT** 윈도우를 관리하고 컨트롤바와 함께 그 위치를 지정합니다. MDI 클라이언트 윈도우는 MDI 차일드 프레임 윈도우의 부모 윈도우입니다.
- MDI 프레임 윈도우는 디폴트 메뉴를 가지고 있어서 활성 차일드 윈도우가 없는 경우 메뉴 바에 표시됩니다. 활성 MDI 차일드 윈도우가 있는 경우 MDI 프레임 윈도우의 메뉴바는 MDI 차일드 윈도우의 메뉴로 교체됩니다.
- MDI 프레임 윈도우는 현재 MDI 차일드 윈도우와 상호작용합니다. 예를 들어 명령 메시지는 MDI 프레임 윈도우가 처리하기 이전에 현재 활성 MDI 차일드 윈도우로 그 처리를 넘깁니다.
- MDI 프레임 윈도우는 차일드 윈도우들을 배열하기 위한 표준 윈도우 명령들을 처리하는 핸들러 함수를 가집니다.
  - **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 있습니다.



ID	Value	Caption
ID_WINDOW_NEW	57643	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

그림. MDI 프레임 윈도우의 디폴트 명령

4-6. CMDIChildWnd 클래스

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

MDI 차일드 윈도우는 프레임 윈도우와 비슷하게 보이지만 몇 가지 차이점이 있습니다. MDI 차일드 윈도우는 데스트탑이 아닌 MDI 프레임 윈도우의 클라이언트 영역 내부에 나타나며, 자신의 메뉴바를 가지지 않고 MDI 프레임 윈도우의 메뉴를 공유합니다. 또한 MFC는 현재의 활성 MDI 차일드 윈도우를 위해서 프레임 윈도우의 메뉴를 자동으로 수정합니다.

CMDIChildWnd 개체가 뷰와 다큐먼트롤 포함하고 있다면 **CFrameWnd** 개체의 경우에서처럼 MFC는 자동적으로 뷰와 다큐먼트를 생성하고 이들을 연결시켜 줍니다.

CMDIChildWnd 클래스는 기본 구현을 **CFrameWnd** 클래스로부터 상속하고 있습니다. 여기에 다음 특성들을 추가적으로 가집니다.

- **CMultiDocTemplate** 클래스와 함께 사용되어 동일한 템플릿에서 생성된 **CMDIChildWnd** 개체는 메뉴를 공유합니다.
- 현재 활성 MDI 차일드 윈도우의 메뉴는 MDI 프레임 윈도우의 메뉴를 대신합니다. 또한 현재 활성 MDI 차일드 윈도우의 타이틀이 프레임 윈도우의 타이틀에 추가됩니다.