

2장 Visual C++와 MFC입문

우리나라에서 일반적으로 많이 사용되었던 C용 컴파일러는 볼랜드 C++와 마이크로소프트의 C컴파일러일 것이다. 그동안 적어도 우리나라에서는 Turbo C, Borland C++같은 볼랜드의 컴파일러가 훨씬 많은 사용자 층을 확보하고 있었다고 생각한다. 볼랜드의 C++컴파일러는 편집기와 디버거를 통합시킨 편리한 통합환경을 제공하고 있었으며, 적은 디스크 용량만으로도 훌륭한 컴파일러를 사용할 수 있었기 때문에 학생들 사이에서 특히 많은 인기를 모았었다. 윈도우3.1의 등장과 함께, 마이크로소프트에서는 윈도우 응용프로그램 개발 도구인 SDK가 발표했는데, 이전의 마이크로소프트 제품이 그랬던 것처럼 사용이 불편하기는 마찬가지였다. 볼랜드는 윈도우 응용프로그램 개발을 위한 C, C++컴파일러를 발표했는데, 도스 환경에서와 비슷한 통합환경을 제공함으로써, 많은 윈도우 응용프로그램 개발자들의 환영을 받았다. 또한 Object Window Library(OWL)라는 C++ 클래스 라이브러리를 내놓아 도스에서의 명성이 윈도우에서도 그대로 이어지는 듯 했다. 이에 비해 마이크로소프트는 OWL에 비해서는 형편없는 수준인 MFC1.0이라는 아주 초라한 클래스 라이브러리 정도만을 겨우 내놓는 정도에 그치고 있었다.

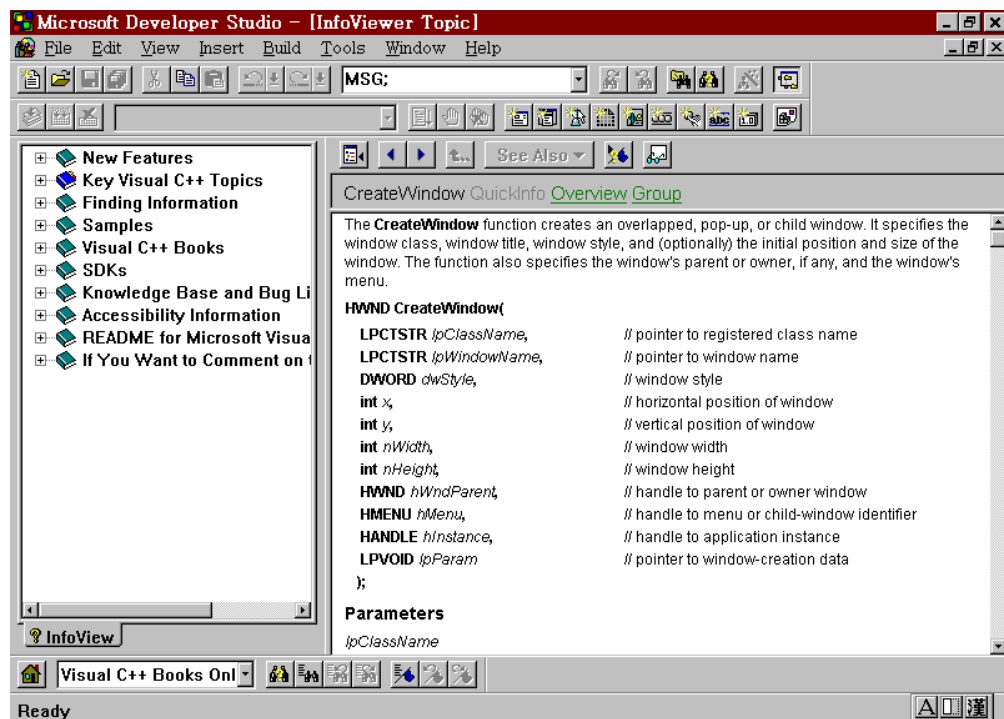
마이크로소프트는 운영체제 개발사라는 입장을 마음껏 이용해, 다른 개발사들보다 한발 앞서 새로운 기술을 채용한 제품을 내놓음으로써 이러한 위치는 역전되는 듯 하다. 클래스 라이브러리로서 구조적으로나 기능적으로는 볼랜드의 OWL이 MFC보다 뛰어나다는 평가도 없지않지만, MFC는 이미 새로운 API 표준처럼 대접받고 있는 것 같다. 또한 근래의 인터넷 열풍에 힘입어 새로운 인터넷 기술에 대한 지원 기능을 재빠르게 컴파일러에 포함시키고, 이에 따라 MFC도 계속 그 판을 올라가면서 개발자들에게 Visual C++, MFC에 대한 선택을 은근히 강요하는 듯하다. 볼랜드도 최근에는 델파이쪽으로 중심을 이동해 Visual Basic과의 한판 대결을 노려보는 것 같아 C++컴파일러 시장에서 Visual C++, MFC가 차지하는 비중은 더욱 커질 것이다.

최근에 볼랜드의 C++ Builder의 평가판이 발표되었는데, 델파이와 거의 똑같은 사용자 인터페이스를 가지면서 C++로 프로그래밍을 할 수 있게 되었다. 델파이가 오브젝트 파스칼을 기초로 하기 때문에 망설였던 많은 C 사용자들에게는 좋은 소식이 아닐 수 없다. 볼랜드의 C++ Builder가 진정한 비주얼 툴로서 성공적으로 자리잡을 수 있을지 기대가 된다.

마이크로소프트사의 윈도우 NT가 서버 시장을 점차 잠식해 감에 따라 클라이언트건 서버건 간에 Win32로 프로그래밍이 가능해 질 것 같다. Win32는 일종의 PC환경에서 표준 API같은 역할을 하게 될 것 같아, 언제나 신기술 습득에 혈떡이고 있는 개발자들로서는 마이크로소프트의 이 달콤한 속삭임에 귀기울이지 않을 수 없게 되었다.

Visual C++

Visual C++ 1.5x까지의 통합환경은 볼랜드 사의 화려한 인터페이스에 비해 사실 좀 조잡한 수준이었다. Visual C++ 통합환경은 Visual C++ 2.0에 들어서면서부터 "Microsoft Developer Studio"란 이름으로 재탄생하게 되었다.



이 개발자 스튜디오는 다른 마이크로소프트의 컴파일러들에도 그대로 사용되는데, C++뿐만, 최근에는 자바개발환경인 마이크로소프트 J++까지 모두 똑같은 통합환경을 사용한다. 마이크로소프트 J++도 설치한 다음 "개발자 스튜디오"를 실행시켜 보면 Visual C++와 똑같은 화면이 뜨는 것을 볼 수 있다. 실제로 실행 파일도 하나이다. 하나의 개발환경을 마이크로소프트 사의 제품들이 모두 사용함으로써, 프로그래밍 언어 별로 별도의 개발 환경을 배워야 하는 부담을 없앴다. 또한 모든 개발환경을 마이크로소프트의 "개발자 스튜디오" 하나로 통합하겠다는 마이크로소프트사의 야심을 보는 듯하다.

마이크로소프트 개발자 스튜디오는 크게 응용프로그램 마법사(AppWizard), 클래스 마법사(ClassWizard), 자원 편집기(Resource Editor)등으로 구성되어 있다.

응용프로그램 마법사(AppWizard)는 일종의 응용프로그램 틀(template) 발생기이다. 새로운 응용프로그램을 만들 때, 일반적으로 필요하게 되는 뼈대를 만들어 주는 것이

다. 윈도 프로그래밍에 입문하는 사람들이 제일 먼저 접하게 되는 황당함이란 화면에 "안녕! 윈도야"라는 문자열을 출력하기 위해 입력해 주어야 하는 그 엄청난(?) 양의 코드다. 도스 시절의 아래와 같은 코드에 익숙해 있는 사람이라면 당연한 일이다.


```
#include <stdio.h>

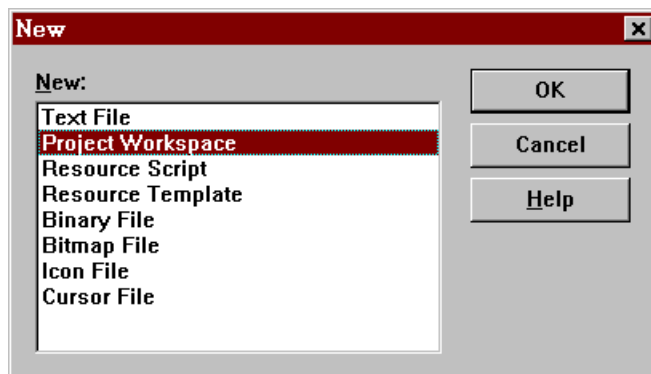
int main(void)
{
    printf("안녕! 윈도야");
    return 0;
}
```

응용프로그램 마법사

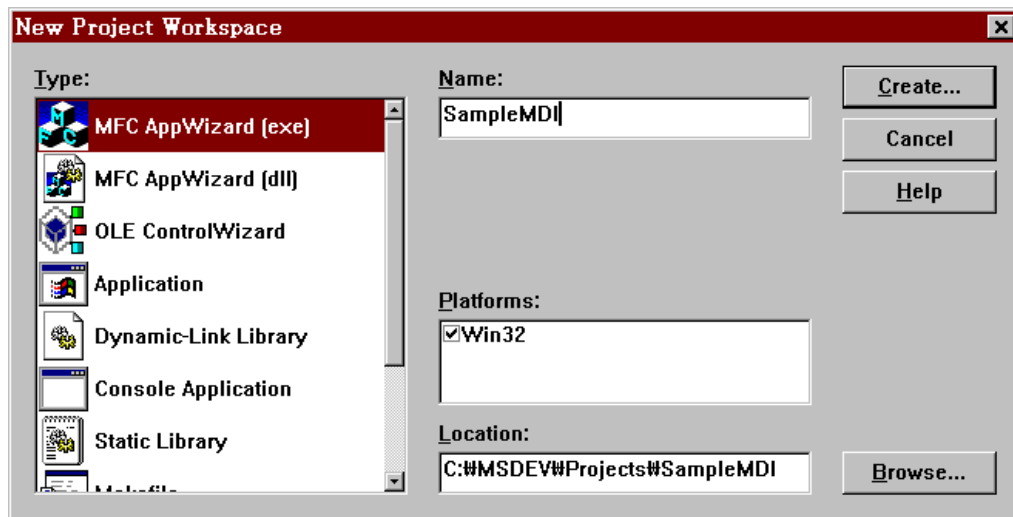
응용프로그램 마법사(AppWizard)에서는 사용자가 자신의 구미에 맞도록 이런 저런 설정을 하고나면, 그에 알맞는 응용프로그램 틀을 자동으로 만들어 준다. 심지어는 친절한 주석문(comment)까지 달아준다. 이렇게 자동으로 생성된 것을 전혀 수정하지 않고도 컴파일, 링크가 되며 실행을 시키면 그럴 듯한 윈도 응용프로그램이 생성된다.

지금, 개발자 스튜디오를 기동시켜 한번 간단한 창이 여러개 열리는 MDI 편집기 프로그램을 만들어보자.

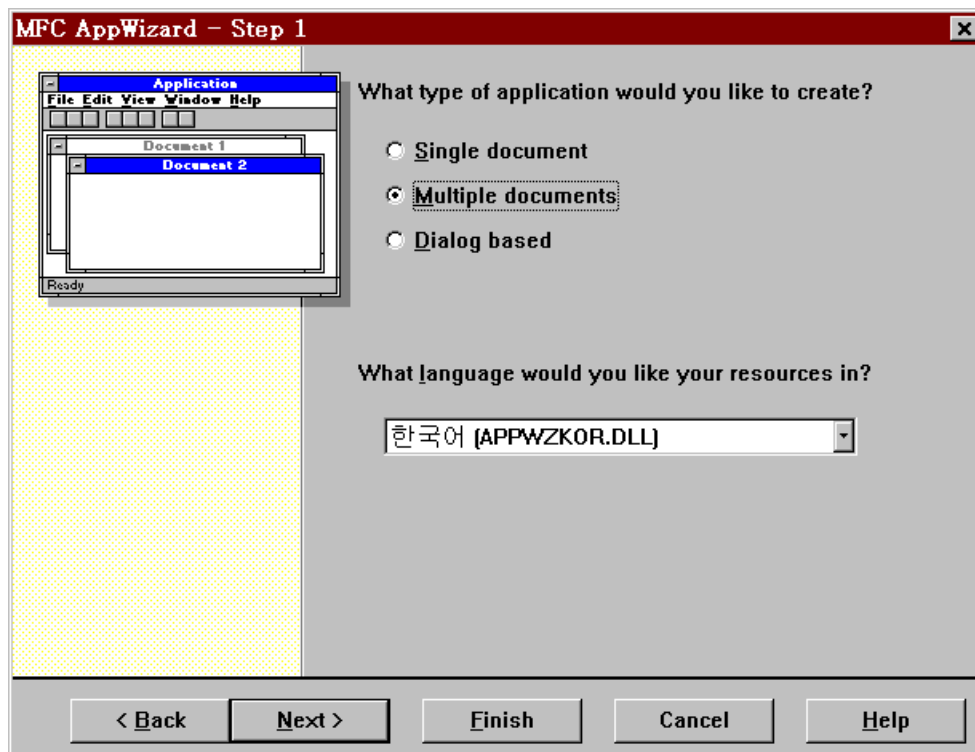
File메뉴에서 New...를 선택한다. 툴바(toolbar)에 있는 을 누르지는 말 것! 이 버튼을 누르면, 그냥 텍스트 파일 편집을 위한 창이 하나 더 열릴 뿐이다.



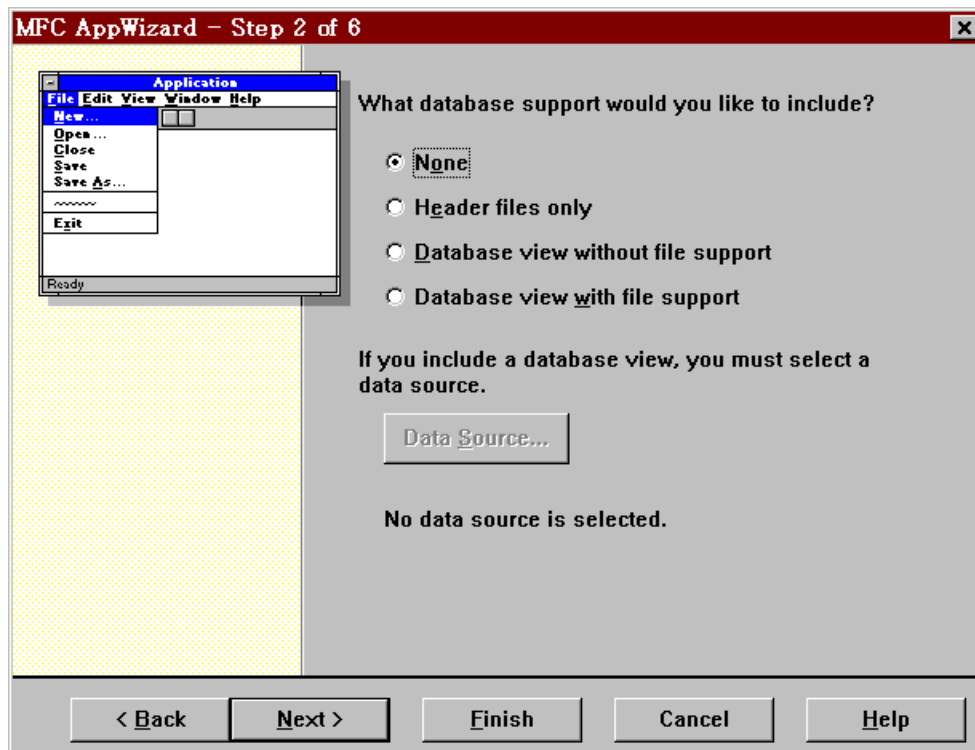
새로운 응용프로그램을 만들려면 이 메뉴에서 "Project Workspace"를 누른다.



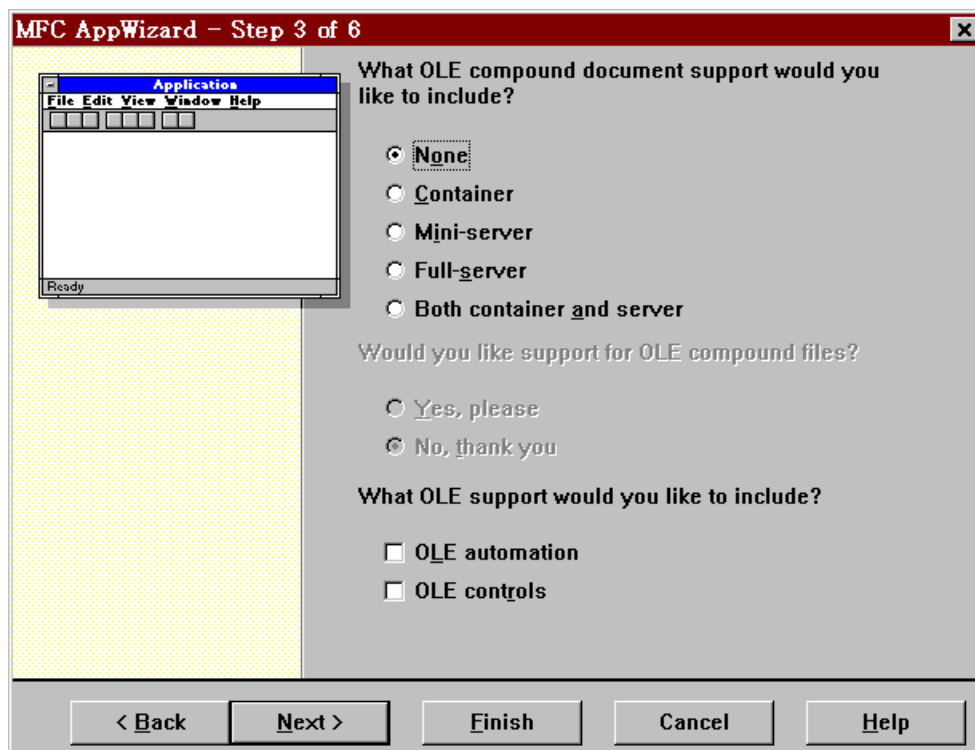
우선 그냥, 제일 위에 있는 "MFC AppWizard(exe)"를 선택하고 "Name:"에 "SampleMDI"라고 입력한 다음 **Create...** 를 누른다.



MDI 응용프로그램을 만들기로 했으므로 **Multiple documents** 를 선택한다. 사용할 언어를 선택하라는 부분에 기쁘게도 "한국어" 항목이 있다. 한글 윈도우95를 사용하고 있다면 기본적으로 "한국어"로 선택된다. 리소스를 한국어로 선택하면 기본적으로 작성되는 메뉴등이 모두 한글로 생성된다. 선택을 마쳤으면, **Next >** 를 누른다.

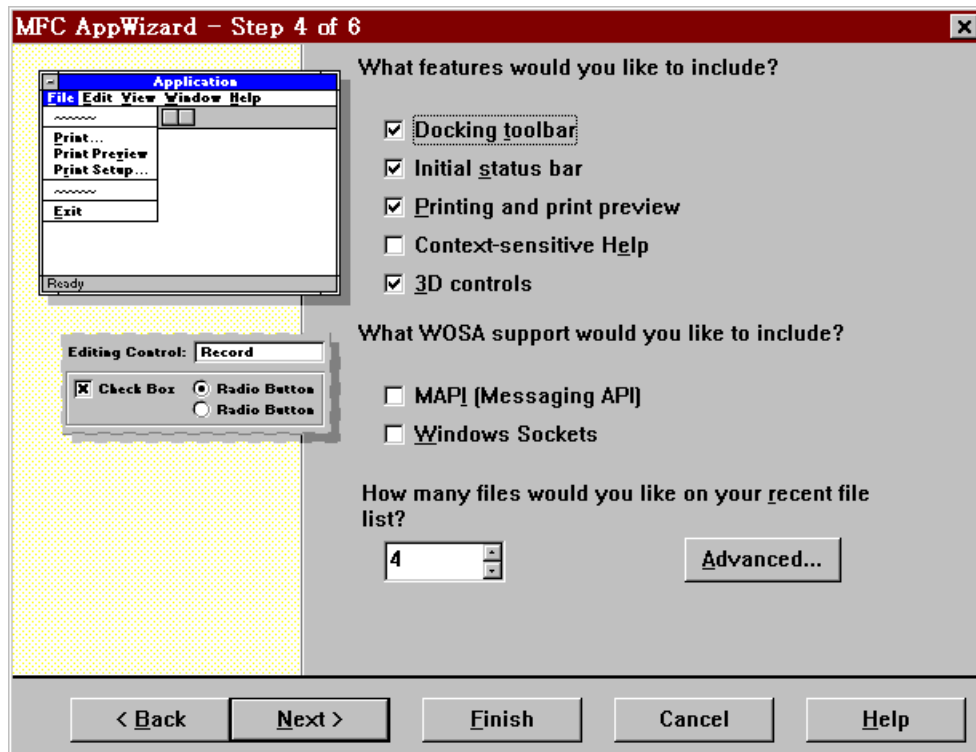


두 번째 단계는 데이터베이스에 대한 선택사항인데, 데이터베이스는 사용하지 않으므로 그냥 **Next >** 를 눌러 다음 단계로 넘어간다.

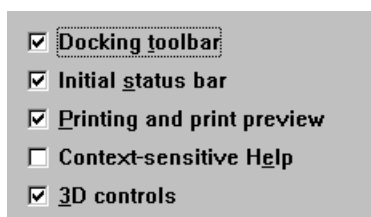


세 번째 단계는 OLE에 대한 선택사항인데, 이 역시 사용하지 않으므로 그냥

Next > 를 눌러 다음 단계로 가자.



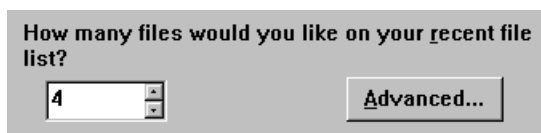
네 번째 단계는 응용프로그램의 겉모습에 대한 것이다.



"Docking toolbar"는 도킹, 즉 우주선이 결합하듯이 툴바가 떠다니가 응용프로그램의 위, 아래, 왼쪽, 오른쪽 아무곳에나 붙을 수 있다는 말이다.

"Status bar"는 대부분의 윈도 응용프로그램에 있는 창 아래부분의 상태 줄을 말한다.

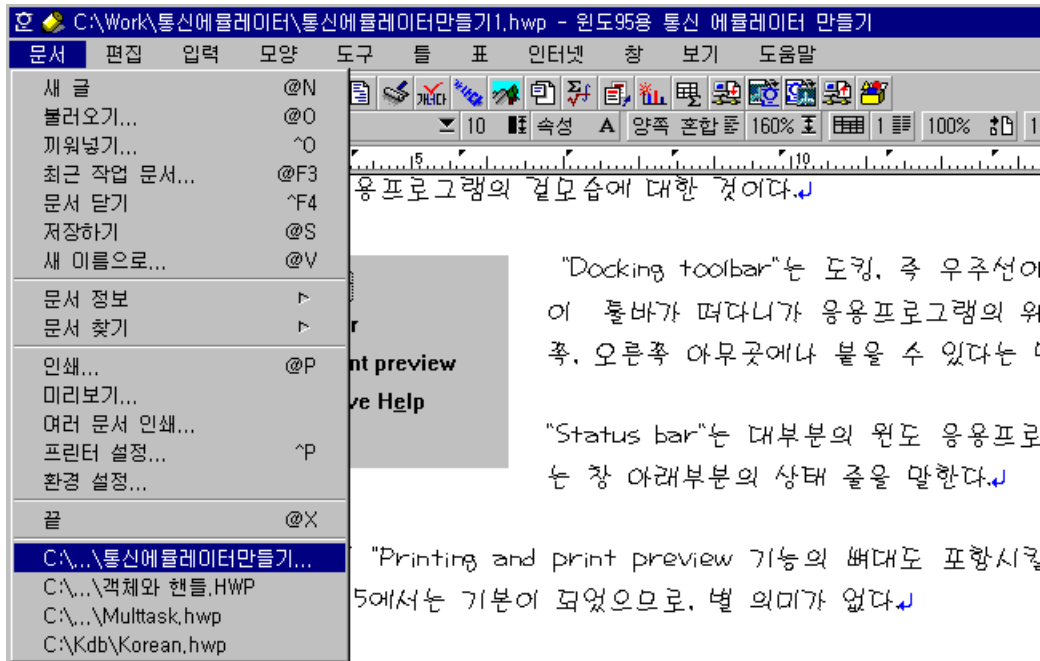
인쇄와 미리보기인 "Printing and print preview" 기능의 뼈대도 포함시킬 수 있다. "3D controls"은 윈95에서는 기본이 되었으므로, 별 의미가 없다.



"recent file list"란 최근에 불러서 작업한 파일 목록이란 뜻으로 이 예에서는 "4"개까지 표시하겠다는 뜻이다. 한글에 "문서"메뉴에 있는 최

근파일 목록의 예를 보면 다음 그림과 같다. 그림의 "문서"메뉴 부분의 아랫부분 쪽을

보면 최근에 작업했던 네 개의 파일이 표시되어 있는 것을 볼 수 있다.

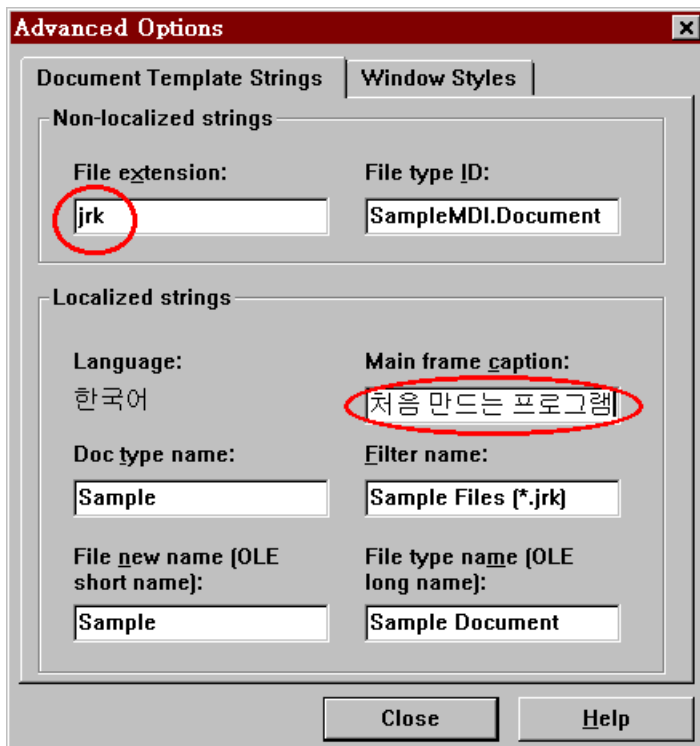


"Docking toolbar"는 도킹, 즉 우주선이 이 툴바가 떠다니가 응용프로그램의 위쪽, 오른쪽 아무곳에나 붙을 수 있다는

"Status bar"는 대부분의 윈도 응용프로그램은 창 아래부분의 상태 줄을 말한다.

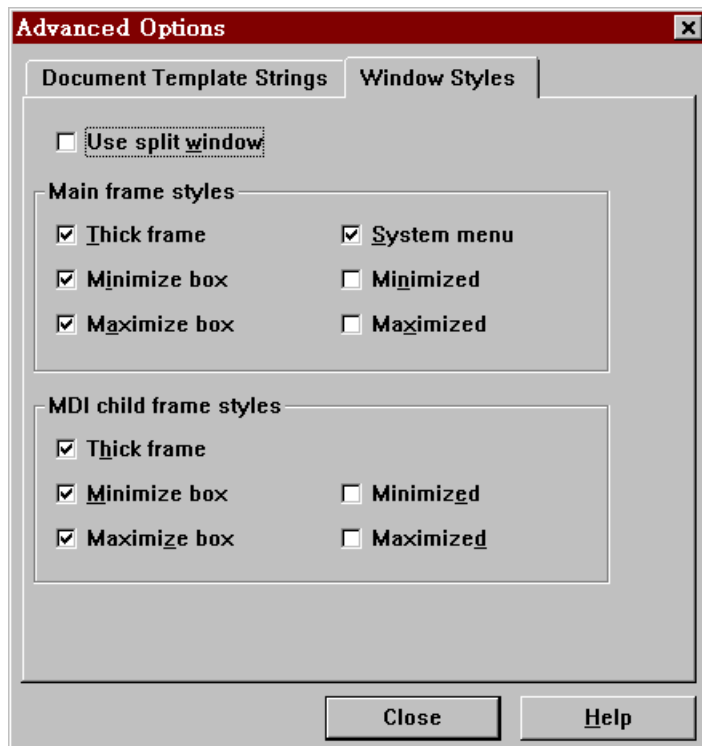
"Printing and print preview 기능의 뼈대도 포함시킴 5에서는 기본이 되었으므로, 별 의미가 없다.

응용프로그램에서 사용할 파일의 기본확장자나 화면에 나타나게 될 창 모양에 대한 설정은 **Advanced...** 버튼을 눌러서 정하게 된다.

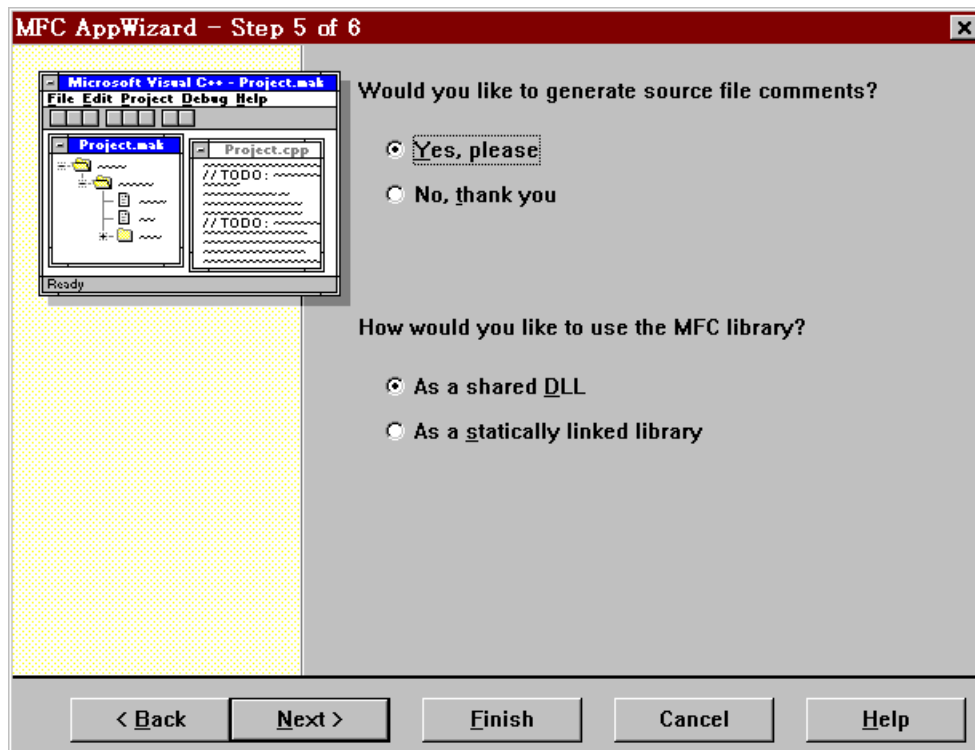


"File extension:" 부분에 응용프로그램에서 사용할 확장자를 지정한다. 연습이니까 자기 이름의 첫 알파벳문자로 한번 지정해 보자. 그러면 불러오기 화면에서 사용하게 될 파일이름 필터도 자동으로 설정되게 된다. 창의 제목으로 쓸 문자도 입력하면 된다.

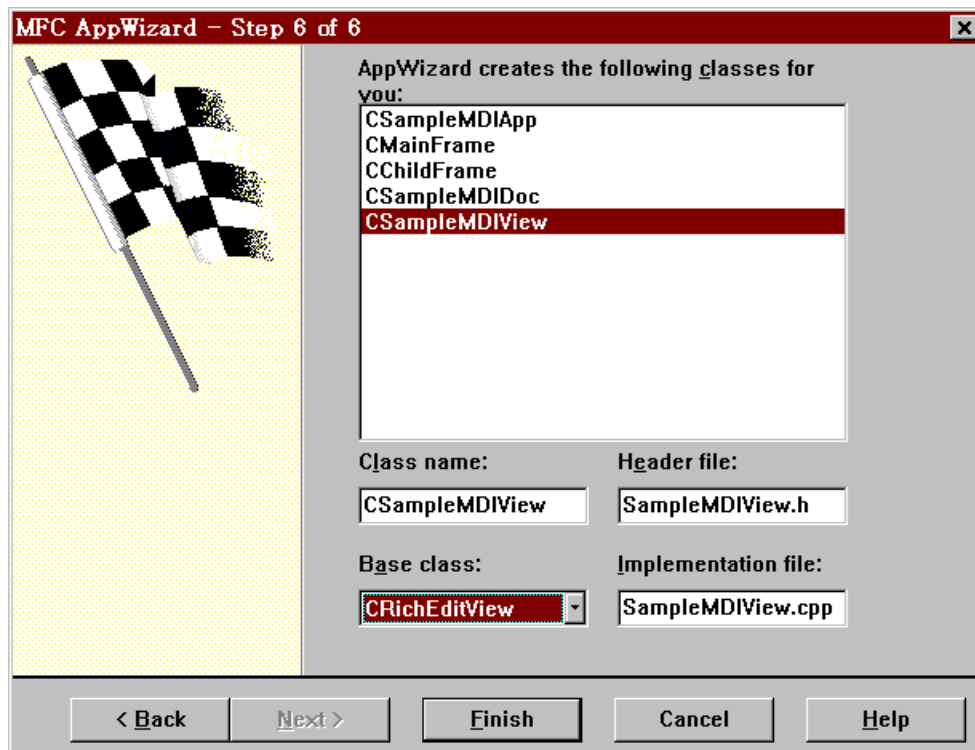
이 대화상자에서 "Window Styles" 탭을 누르면 창의 모양을 설정할 수 있는 다음과 같은 대화상자가 나타난다.



원하는 모양대로 설정이 끝났다면 **Close** 버튼을 누른 다음, **Next >** 버튼을 눌러 다음 단계로 진행하자.



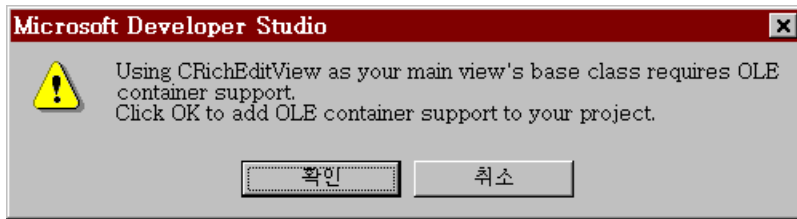
다섯 번째 대화상자는 소스코드에 주석을 달 것인지와 MFC라이브러리를 DLL로 사용할 것인지, 아니면 실행파일에 포함시켜 사용할 것인지를 결정하는 것인데, 둘 모두 설정값을 바꾸지 않고 그대로 두는 것이 좋다. **Next >** 를 눌러 마지막 단계인 다음 여섯 번째 단계로 진행해 보자.



마지막 단계인 여섯 번째 단계는 별다른 것 없이 그냥 **Finish** 버튼만 눈에 크게 들어온다. 나중에 MFC 프로그래밍에서 다루겠지만, MFC 프로그래밍은 기본적으로 내용-보기(Document-View) 구조로 되어있다. 내용(Document)부분에는 프로그램에서 사용되는 것가지 변수들이나 자료구조가, 보기(View)부분에는 이 내용들을 어떻게 보여주고, 사용자의 입력에 어떻게 반응할 지에 대한 코드가 나누어 담기는 것이 보통이다. 여기서는 그렇다는 정도만 살펴보고 넘어가자.

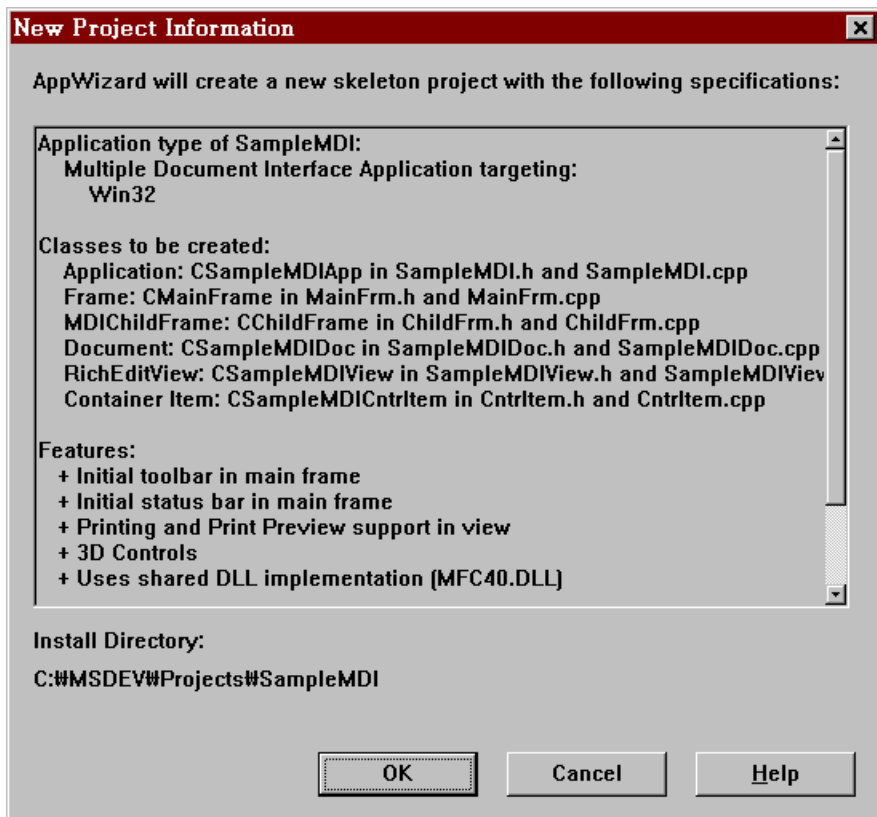
처음에 MDI 편집기 프로그램을 만들어 보기로 했는데, 지금까지 이전의 다섯 단계에서는 편집기 프로그램을 만든다는 것을 설정한 적이 없다. 그렇다면... 혹시 이 응용 프로그램 마법사는 진짜 뼈대(?)만 만들고, 나머지 편집부분은 직접 코딩...하고 걱정할지 모르겠다. 비밀은 마지막 단계에 숨어 있다. 위의 대화상자를 보면, 만들어질 다섯 개의 클래스들이 있는데, 이들을 하나씩 클릭해보면, "Base class:" 항목이 선택 가능하게 되는 클래스가 하나 있다. 바로 CSampleMDIView 클래스이다. 여러 가지 항목 중에서 CRichEditView를 선택해 보자. CRichEditView는 윈95에 포함되어 있는 새로운 문서편집기인 워드패드에서 사용할 수 있는 문서 장식을 사용할 수 있는 편집기 클래스이다. 단순한 텍스트 편집이 아니라, 문자크기, 모양 조정과 문단 모양 조정등이 가능하다는 얘기이다. 정말일까? 떨리는 손으로 **Finish** 버튼을 눌러 확인해 보자.

그런데, 이게 웬일?



한번 "잘 읽어보라"는 느낌표(!) 대화상자가 나타난다. 잘 읽어보니 응용프로그램에서 사용하는 뷰가 CRichEditView를 사용하려면 OLE컨테이너를 지원하도록 해야한다는 의미인데, 처음에는 잘 모르는 얘기니 그냥 "확인"을 과감하게 눌러본다.

그런데, 또 ...



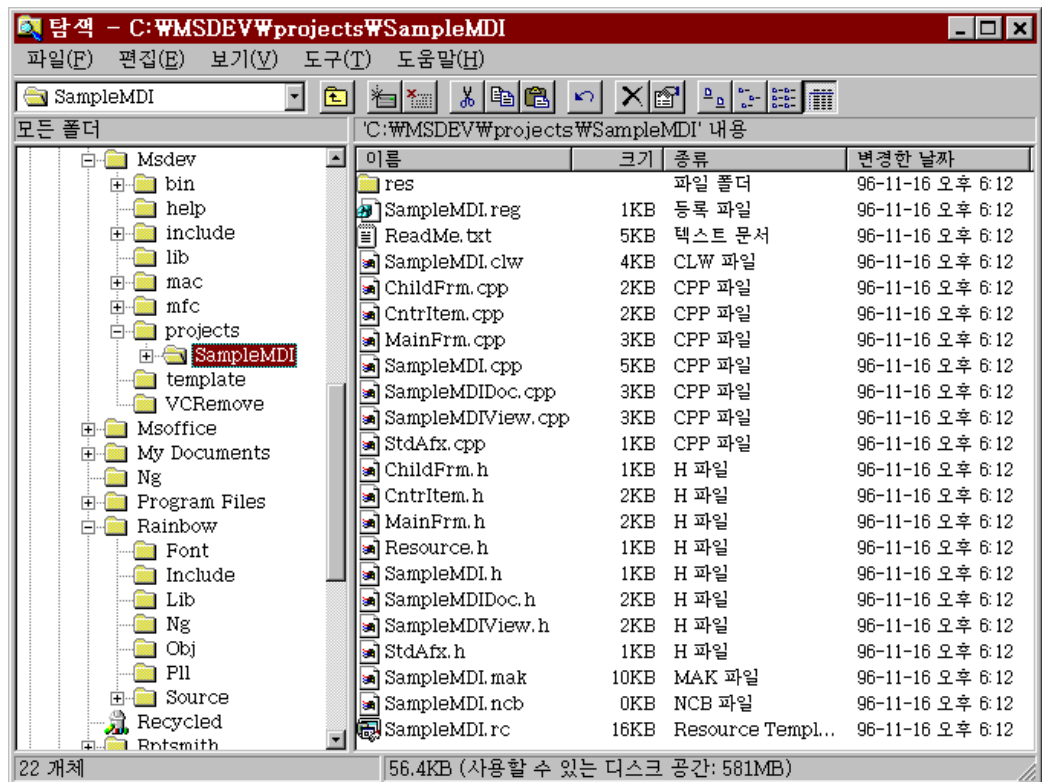
하지만, 뭐 긴장할 필요는 없다. 지금까지 자신이 선택한 결정사항과 그에 따라 생성되게 될 응용프로그램에 대한 요약을 보여주는 친절한 대화상자니까. 이번은 진짜 마지막이다. **OK**를 눌러 결과를 보도록 하자.

새로운 응용프로그램을 만들고 있다는 대화상자가 잠깐 나타났다가 금방사라지고 나면, 화면에는 별다른 변화가 없어보인다.



자세히 살펴보니 창의 제목이 바뀌고 화면 왼쪽 편의 창에 "SampleMDI classes"라는 것이 생겼다. 그렇다면 실제로 생긴 파일들은 어디에 얼마나 생겼는지 확인해 보자.

탐색기를 열어 MSDEV\project\폴더로 이동해 보자



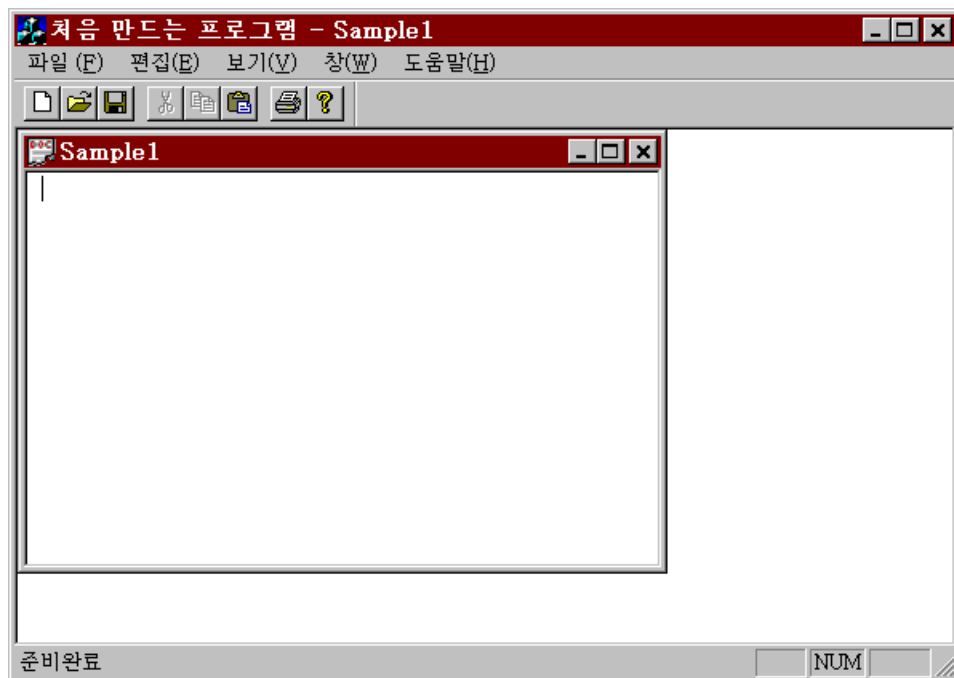
res폴더를 제외하고도 무려 21개의 파일이 생겼다.

각 파일의 내용은 조금 후에 살펴보기로 하고, 지금 만들어진 파일들이 어떻게 동작할 지 우선 먼저 컴파일부터 해보자. "Build"메뉴의 "Build SampleMDI.exe"항목을 선택한다.

```
-----Configuration: SampleMDI - Win32 Debug-----
Compiling resources...
Compiling...
StdAfx.cpp
Compiling...
CntrItem.cpp
SampleMDIView.cpp
```

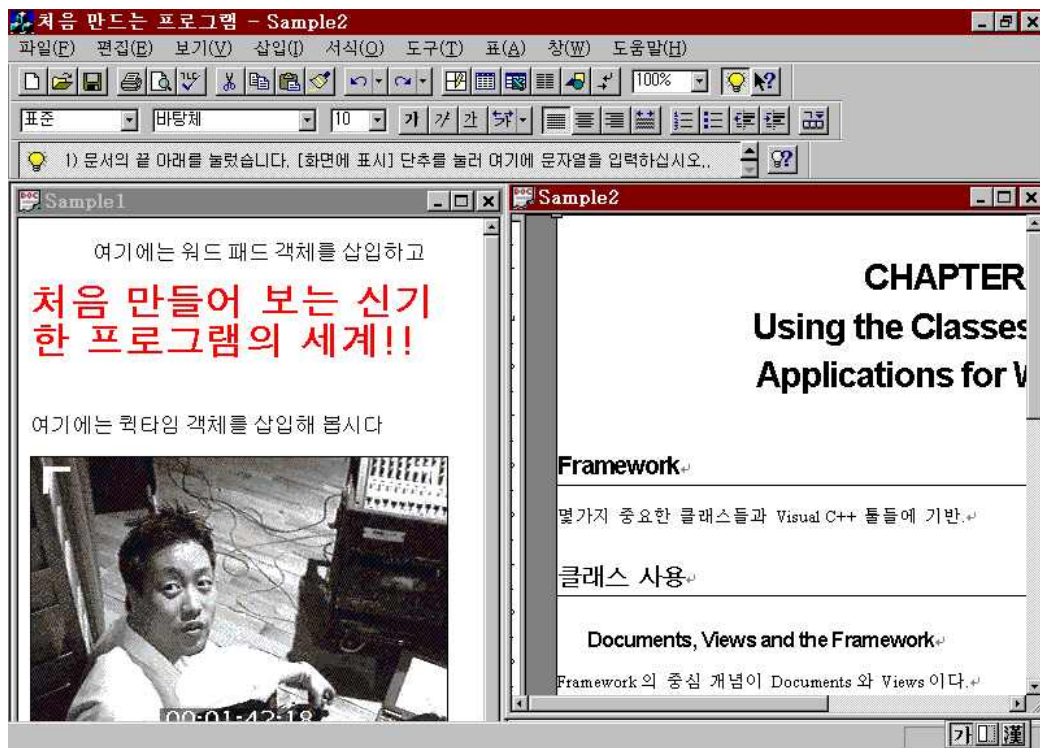
```
SampleMDIDoc.cpp
ChildFrm.cpp
MainFrm.cpp
SampleMDI.cpp
Generating Code...
Linking...
SampleMDI.exe - 0 error(s), 0 warning(s)
```

컴파일과 링크가 무사히 끝났으니 설레이는 마음으로 "Build"메뉴의 "Execute SampleMDI.exe"항목을 선택하면 잠시 후...



위 그림과 같은 완성된 프로그램이 나타난다. 이 프로그램은 "파일"메뉴의 "열기...", "저장...", 등의 기능이 완벽하게 수행되며, "미리보기"와 "인쇄하기" 기능도 완벽하진 않지만 그런대로 수행되는 완전한 형태의 하나의 응용프로그램이다. 한 줄의 추가 코드 없이도 이 정도의 응용프로그램을 얻을 수 있다는 점에 너무 감격해 하진 말 것!, 왜냐하면 만들어진 코드의 어느 부분에 자신의 코드를 추가해야 하는지를 완전히 소화하려면 아직도 넘어야 할 험한 길이 너무나 많기 때문이다.

단순한 MDI 문서 편집기처럼 보이지만 "편집"메뉴에 있는 "새 개체 삽입"을 통해, 다른 OLE개체들을 포함시킴으로써, 코딩 한 줄 없이 다음과 같은 프로그램을 얻을 수 있다. OLE2.0의 시각 편집(Visual Editing)기능을 통해 MS워드 개체를 포함시키면 이 프로그램 자체가 MS워드가 된 듯한 모습을 보여준다.



메뉴 부분이 MS워드로 바뀐 것을 볼 수 있다.

맛보기는 이 정도로 하고, 자동으로 생성된 파일들의 종류와 그 용도를 살펴해보도록 하자.

먼저, 열어 볼 파일은 ReadMe.txt이다. 이 파일에는 자동 생성된 파일의 종류와 각 파일의 용도에 대한 설명이 나와 있다.

```
=====
MICROSOFT FOUNDATION CLASS LIBRARY : SampleMDI
=====
```

AppWizard has created this SampleMDI application for you. This application not only demonstrates the basics of using the Microsoft Foundation classes but is also a starting point for writing your application.

This file contains a summary of what you will find in each of the files that make up your SampleMDI application.

SampleMDI.h

This is the main header file for the application. It includes other project specific headers (including Resource.h) and declares the CSampleMDIApp application class.

SampleMDI.cpp

This is the main application source file that contains the application class CSampleMDIApp.

SampleMDI.rc

This is a listing of all of the Microsoft Windows resources that the program uses. It includes the icons, bitmaps, and cursors that are stored in the RES subdirectory. This file can be directly edited in Microsoft Developer Studio.

res\SampleMDI.ico

This is an icon file, which is used as the application's icon. This icon is included by the main resource file SampleMDI.rc.

res\SampleMDI.rc2

This file contains resources that are not edited by Microsoft Developer Studio. You should place all resources not editable by the resource editor in this file.

SampleMDI.reg

This is an example .REG file that shows you the kind of registration settings the framework will set for you. You can use this as a .REG file to go along with your application or just delete it and rely on the default RegisterShellFileTypes registration.

SampleMDI.clw

This file contains information used by ClassWizard to edit existing classes or add new classes. ClassWizard also uses this file to store information needed to create and edit message maps and dialog data maps and to create prototype member functions.

////////////////////////////////////

For the main frame window:

MainFrm.h, MainFrm.cpp

These files contain the frame class CMainFrame, which is derived from CMDIFrameWnd and controls all MDI frame features.

res\Toolbar.bmp

This bitmap file is used to create tiled images for the toolbar. The initial toolbar and status bar are constructed in the CMainFrame class. Edit this toolbar bitmap along with the array in MainFrm.cpp to add more toolbar buttons.

////////////////////////////////////

AppWizard creates one document type and one view:

SampleMDIDoc.h, SampleMDIDoc.cpp - the document

These files contain your CSampleMDIDoc class. Edit these files to add your special document data and to implement file saving and loading (via CSampleMDIDoc::Serialize).

SampleMDIView.h, SampleMDIView.cpp - the view of the document

These files contain your CSampleMDIView class. CSampleMDIView objects are used to view CSampleMDIDoc objects.

res\SampleMDIDoc.ico

This is an icon file, which is used as the icon for MDI child windows for the CSampleMDIDoc class. This icon is included by the main resource file SampleMDI.rc.

```

////////////////////////////////////.

AppWizard has also created classes specific to OLE

CntrlItem.h, CntrlItem.cpp - this class is used to
manipulate OLE objects. They are usually displayed by your
CSampleMDIView class and serialized as part of your CSampleMDIDoc class.

////////////////////////////////////.
Other standard files:

StdAfx.h, StdAfx.cpp
These files are used to build a precompiled header (PCH) file
named SampleMDI.pch and a precompiled types file named StdAfx.obj.

Resource.h
This is the standard header file, which defines new resource IDs.
Microsoft Developer Studio reads and updates this file.

////////////////////////////////////.
Other notes:

AppWizard uses "TODO:" to indicate parts of the source code you
should add to or customize.

If your application uses MFC in a shared DLL, and your application is
in a language other than the operating system's current language, you
will need to copy the corresponding localized resources MFC40XXX.DLL
from the Microsoft Visual C++ CD-ROM onto the system or system32 directory,
and rename it to be MFCLOC.DLL. ("XXX" stands for the language abbreviation.
For example, MFC40DEU.DLL contains resources translated to German.) If you
don't do this, some of the UI elements of your application will remain in the
language of the operating system.

////////////////////////////////////.

```

SampleMDI.rc

프로그램에서 사용되는 리소스들이다. 아이콘, 비트맵, 커서 등은 res라는 부 디렉토리에 들어 있다. 이 파일들이 마이크로소프트 개발자 스튜디오에서 직접 편집도 할 수 있다.

CntrlItem.h, CntrlItem.cpp – OLE 구현

OLE객체를 다루기 위해 필요하다.

Resource.h

자원에 대해 ID를 정의를 해 둔 파일. 사용자가 원하는 이름으로 바꿔 쓸 수도 있다.

SampleMDI.reg

윈도3.1에서는 INI파일에 각종 정보를 저장했지만, 윈도95는 레지스트리라는 것을 사용하는데, 이 파일은 레지스트리에 저장할 이 응용프로그램의 정보에 대한 예를 보인 것이다.

SampleMDI.clw

마이크로소프트 개발자 스튜디오의 클래스 마법사에서 사용하는 정보를 저장하는 파일이다. 클래스 마법사는 이 파일에 클래스들을 편집하거나 새로운 클래스를 추가하게 되면 그 정보를 저장하게 된다. 사용자가 편집하는 일은 거의 없다.

Stdafx.h, Stdafx.cpp

MFC로 만들어진 프로그램을 컴파일하면 상당히 MFC 헤더 파일들을 매번 컴파일해야 하는데, 이 때 시간이 꽤 걸리게 된다. 이 파일들은 사실, 사용자가 거의 수정하지 않는 내용이기 때문에 처음에 한번 컴파일하면, 그 다음부터는 다시 컴파일할 필요가 없는 내용들이다. 이런 불편을 해소하기 위해 미리 컴파일(pre-compile)을 해 두는 방법이 이용되는데(precompiled header : PCH), 이 두 파일은 이 미리 컴파일을 위해 필요한 파일이다. Stdafx.cpp파일을 한번 열어보면, 그냥 `#include "stdafx.h"` 한 줄만 이 달랑 들어 있을 뿐이다.

SampleMDI.h, SampleMDI.cpp – 응용프로그램

이 응용프로그램 전체에 걸쳐 사용될 주 헤더 파일과 소스 파일이다. 헤더 파일에는 다른 곳에서 사용되는 헤더들이 정의되어 있고, 응용프로그램 클래스인

CSampleMDIApp가 선언되어 있다. 소스 파일은 CSampleMDIApp 클래스가 포함되어 있는 이 응용프로그램의 메인 모듈이다.

MainFrm.h, MainFrm.cpp – 메인 프레임

MDI 프레임을 다룰 수 있는 CMDIFrameWnd를 상속받은 CMainFrame에 대한 파일이다.

SampleMDIDoc.h, SampleMDIDoc.cpp – 내용

내용-보기 구조는 흔히 다큐먼트-뷰 구조라고 부르는데, 다큐먼트라는 용어가 '문서, 서류'라는 의미에 더 익숙해 있기 때문에 그 의미를 선뜻 이해하기 어렵다는 생각이 들어서 '내용'이란 말로 바꿔 보았다. 나중에 자세히 다루겠지만, 아무래도 '내용'이란 용어가 더 적절하다고 고개를 끄덕이게 될 것이다. 일단은 기본 구조가 내용-보기 구조로 되어 있다는 것만 기억해 두자.

이 파일들은 내용과 보기 구조에서 내용(Document) 부분을 구현한다. 파일 저장, 불러오기를 구현하거나 내부적으로 사용되는 데이터를 추가하고 싶다면 이 파일에 한다.

SampleMDIView.h, SampleMDIView.cpp – 보기

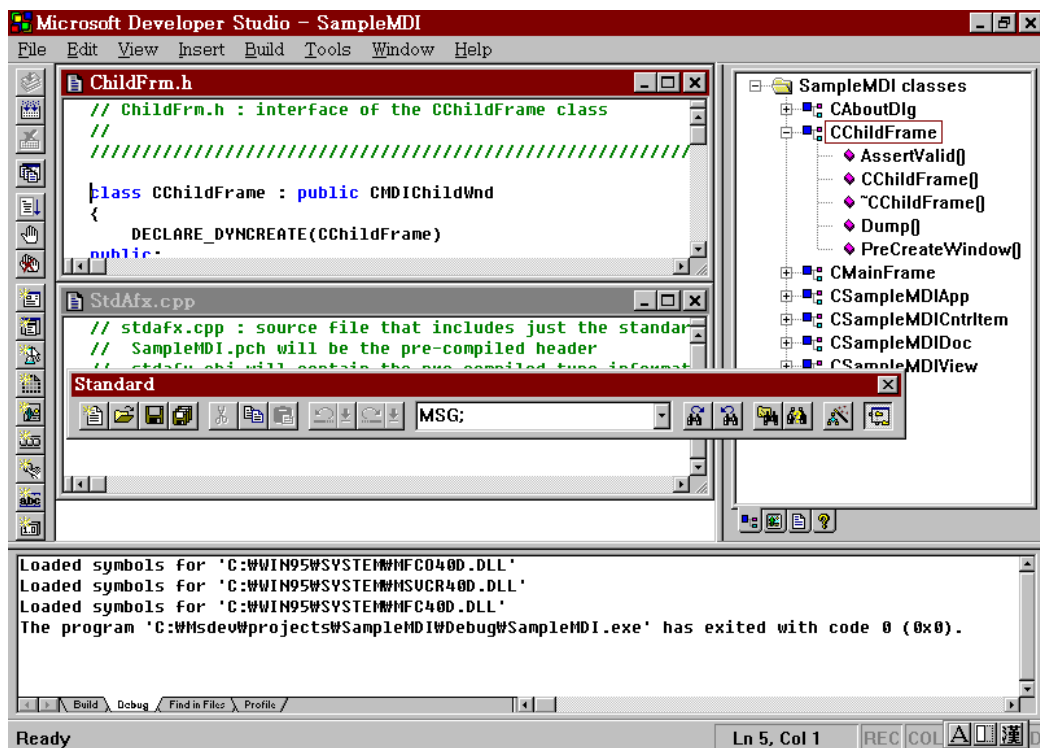
내용과 보기 구조에서 보기(View) 부분을 구현한다. 일반적으로 수정해야 하는 파일은 내용-보기 구조 관련 파일(SampleMDIDoc.h, SampleMDIDoc.cpp, SampleMDIView.h, SampleMDIView.cpp)과 메인 프레임(MainFrm.h, MainFrm.cpp), 그리고 응용 프로그램(SampleMDI.h, SampleMDI.cpp) 부분의 파일이다. Stdafx.h도 수정해야 하는 경우가 있지만, 처음에는 거의 수정할 일이 없다. 자원 편집에 관련된 파일들은 마이크로소프트 개발자 스튜디오 내에서 시각적으로(Visual) 편집을 해서 저장하면, 알맞게 수정되므로 직접 파일을 열어서 편집해야 할 일은 거의 없다. 소스 코드 수정과 편집에는 클래스 마법사의 도움을 받으면 훨씬 손이 덜가게 일할 수 있다.

각 부분의 명칭

마이크로소프트 개발자 스튜디오의 각 화면 구성 요소들은 모두 Docking이 가능한

것들로서, 마우스로 끌어 다른 곳에 두면, 별도의 창으로 열리거나 또는 다른 위치에 붙게 된다. 일반적인 화면을 다음과 같이 구성할 수도 있다.

마이크로소프트 개발자 스튜디오에서는 이전의 make파일 대신에 mdp파일을 사용하는데, 이것은 이전의 make파일이 하나의 프로그램을 생성하기 위해 사용되는데 대해 여러개의 프로그램을 생성할 수 있는 하나 이상의 make파일로 구성되어 있다고 생각하면 된다. 그리고, 이전과는 달리 make파일 대신 워크스페이스라는 용어를 사용한다. "File"메뉴의 "Open Workspace..."는 이전 버전의 make파일 열기 줌으로 생각하면 된다. 마이크로소프트 개발자 스튜디오에서는 새로운 응용프로그램 하나를 만드는 것을 하나의 프로젝트라고 부른다. 새로 생성된 일련의 파일이나 클래스를 위한 작업공간도 "프로젝트 워크스페이스"라고 부른다.

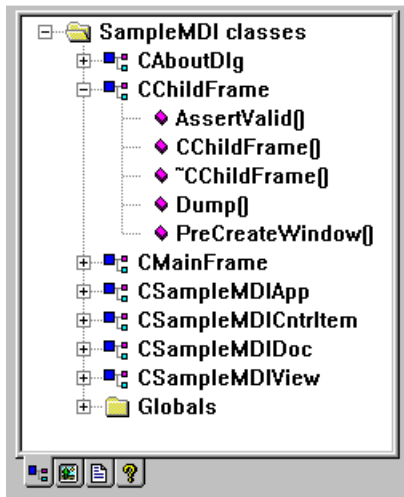


프로젝트 워크스페이스

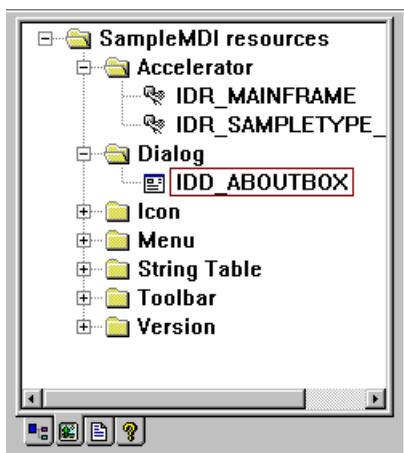
화면의 각 구성 요소를 살펴보자. 먼저 이전의 Visual C++와 크게 달라진 것이 바로 프로젝트 워크스페이스이다.

화면 아래 부분에 네 개의 탭이 보이는데, 왼쪽부터 차례로, 클래스 뷰(class view), 리소스 뷰(resource view), 파일 뷰(file view), 정보 뷰(info view)라고 불린다. 현재

나타난 그림은 바로 클래스 뷰이다.

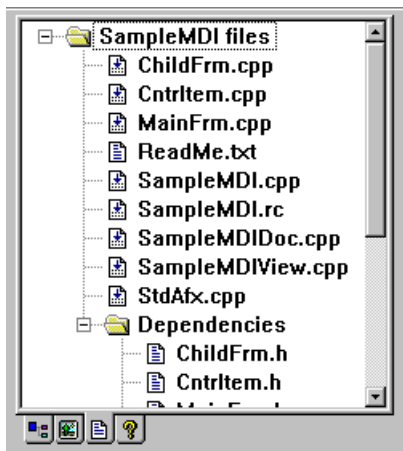


클래스 뷰에서는 프로그램에서 사용되는 클래스들을 한눈에 볼 수 있다. 클래스 이름을 더블클릭하면 클래스가 선언되어 있는 헤더파일이 열리게 된다. 클래스 이름 앞에 있는 '+'기호를 클릭해서 나오는 클래스의 멤버를 클릭하면, 해당 멤버함수가 정의된 .cpp파일이 열리고, 화면에는 바로 그 멤버함수가 보이게 된다.

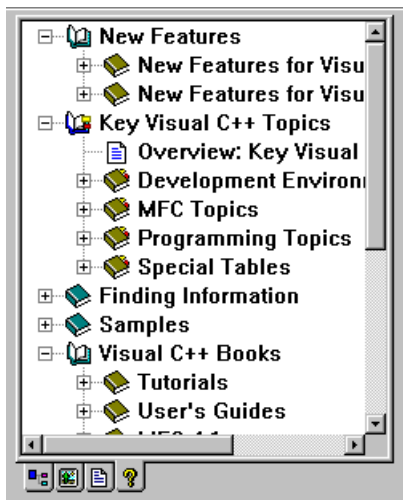


리소스 뷰는 Visual C++ 1.5x에서는 분리된 리소스 편집기를 통합 환경안에 포함시킨 것으로, 그림을 살펴보면 알겠지만, 자동으로 생성된 리소스에는 액셀러레이터, 대화상자, 아이콘, 메뉴, 문자열 테이블, 도구상자, 버전 정보가 있다. 리소스를 더블클릭하면 해당 리소스를 편집할 수 있다.

리소스 뷰 부분의 각 리소스를 모두 열어보면 액셀러레이터와 아이콘과 메뉴와 도구상자가 모두 IDR_MAINFRAME이란 같은 이름을 가진 것을 발견할 수 있는데, 이렇게 같은 이름을 여러 리소스가 갖고 있는 이유는 나중에 살펴보기로 하고, 일단 액셀러레이터, 아이콘, 메뉴, 도구상자 이 네 개의 리소스의 ID가 같아야 한다는 점만 알고 넘어가도록 하자.



다음으로 파일 뷰는 이 응용프로그램에 사용되는 파일들이 모두 나열되어 있는데, 역시 파일이름을 더블클릭하면 편집 상태로 들어갈 수 있다. 원하지 않는 파일은 파일 선택 후에 Del키를 눌러 이 프로젝트 파일에서 지울 수도 있다. 하지만, 그렇다고 실제 물리적으로 디스크에 있던 파일이 지워진 것은 아니다. 단지 이 프로젝트에서 사용하지 않겠다는 의미일 뿐이다. 새로 파일을 추가하려면 "Insert"메뉴에서 "Files into project..."를 선택해서 원하는 파일을 고르면 된다.



마지막으로 정보 뷰는 도움말을 위한 것인데, 책과 같은 개념으로 사용이 아주 편리하고도 쉽다. 물론 찾고자 하는 항목을 찾기에 아주 쉽게 되어 있다. 원하는 부분만 따로 인쇄해서 볼 수도 있으며, 소스코드의 함수 부분에서 F1을 눌러 도움말을 볼 때, 현재 도움말이 전체 도움말 책에서 어떤 부분에 해당하는지도 쉽게 볼 수 있다.

프로젝트

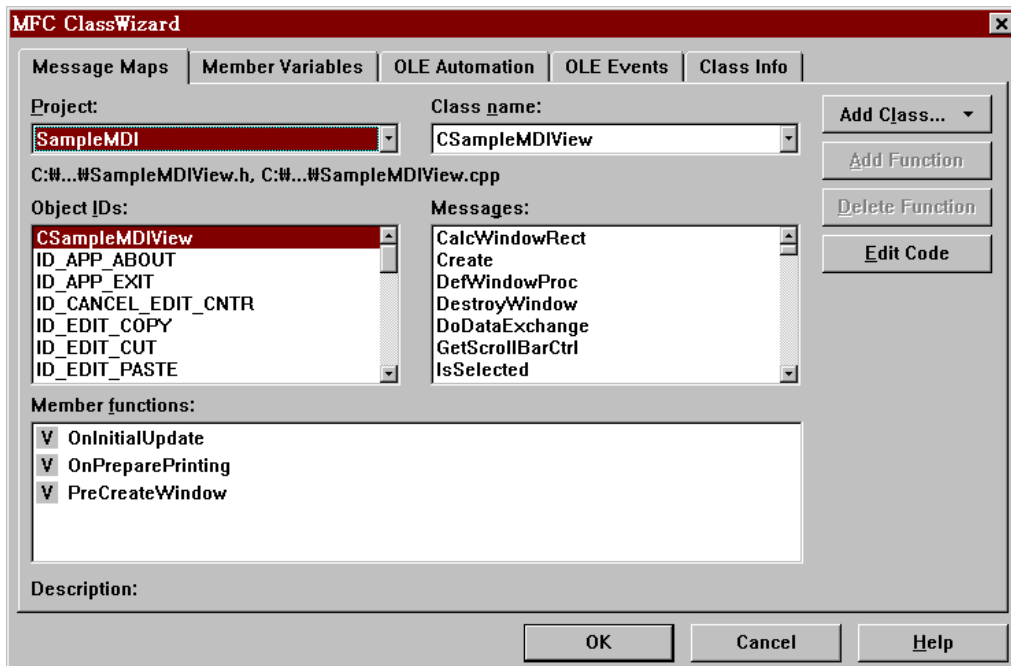
Visual C++ 1.5x를 써 본 사람이라면, 누구나 불편을 느끼던 점으로, 디버그 판과 배포용 판의 관리 문제이다. 개발 단계에서 디버그 상태로 컴파일, 링크를 하다가 최종 단계에서 배포용(Release)으로 다시 컴파일, 링크를 하면 이전의 디버그 판의 실행 파일은 배포용 실행 파일이 덮어써 버리게 된다는 점 때문에 여러 가지 번거로운 일이 많았다.



Visual C++ 4.x의 개발자 스튜디오는 프로젝트 창에서 "SampleMDI-Win32 Debug"와 "SampleMDI-Win32 Release"를 선택할 수 있으며, 이들은 별도의 "Debug", "Release"폴더로 나누어져 파일들이 생성되게 된다. 이로써 디버그 판은 "Debug"폴더에 생성되고 배포판은 "Release"폴더에 생성되기 때문에 관리가 쉬워졌다.

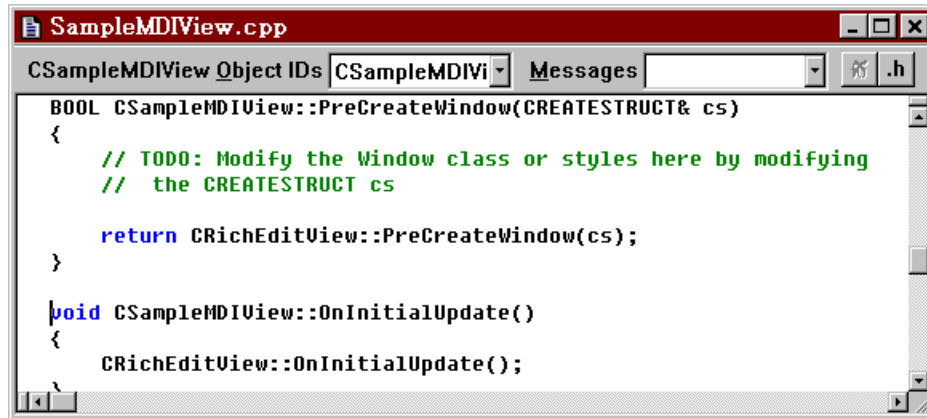
클래스 마법사

클래스를 새로 추가하거나 삭제하는 것뿐만 아니라, 클래스 내의 새로운 멤버함수나 멤버변수의 추가, 삭제, 편집에도 사용된다. 상당한 분량의 코드를 자동으로 생성해 주며, 또한 친절하게도 새로 만들어진 클래스의 멤버함수의 어느 부분에 사용자가 코드를 작성해야 하는지도 가르쳐 준다. 아마도 앞으로 가장 많이 그리고 가장 자주 사용하게 될 것이다.



뿐만 아니라, 소스 코드 편집 상에서도 직접 새로운 멤버 함수를 추가할 수도 있고

록 되어 있다. 코드 편집 창 윗부분의 Object IDs와 Messages부분은 클래스 마법사의 그것과 기능도 같고, 의미 또한 같다.



오른쪽 부분의 **.h**를 누르면, 이 .cpp파일의 헤더 파일을 바로 열어 볼 수 있어서 매우 편리하다.

이 정도의 지식만 있어도 이제 충분히 Visual C++를 사용할 준비가 되어 있다고 해도 되겠다. 물론 약간(?)의 윈도우 프로그래밍 사전 지식과 C++에 대한 약간(?)의 지식이 필요하겠지만 말이다.

코딩 한 줄 없이 훌륭한 윈도우 응용프로그램을 만들어 본 여러분은 의기 양양하겠지만, 사태는 그렇게 단순하지만은 않다. 자동으로 생성된 소스코드는 MFC라는 물건을 살펴보지 않고서는 그냥 단순한 암호 수준의 문장일 뿐이다. 생성된 소스코드에 한 줄이라도 자신의 코드를 추가하려면 실로 엄청난(?) 노력이 추가로 필요하게 되는 것이다. 이 소스 코드의 구조에 대해 알려면 먼저 MFC라는 높은 산에 올라야 한다. MFC가 제시하는 장미빛 환상들 중 일부만이 사실이더라도 충분히 MFC를 배울 용기를 낼 만큼 근래 MFC가 일종의 윈도우 표준 C++ API쯤으로 정착되는 것 같다.

API프로그래밍과 MFC프로그래밍

맛보기로 만들어 본 응용프로그램의 막강한 기능을 본 사람은 윈도우 응용프로그램을 처음 접하면서 "안녕? 세상아!!"라는 문자열을 화면에 뿌리기 위해 얼마나 많은 코드를 직접 입력했는가를 생각하면 실로 감개가 무량할 것이다.

MFC는 Microsoft Foundation Class의 머릿글자이다. 그 구조상의 여러 가지 문제에도 불구하고 근래에 가장 인기 있는 윈도우용 클래스 라이브러리가 아닐까 생각한다. 클래스 라이브러리가 그러하듯이 MFC는 윈도우 API로 프로그래밍할 때 해 주었던 수많은 자질구레한 것들은 손쉽게 해 주는 장점을 가지고 있다.

MFC(Microsoft Foundation Class)

현재 가장 최신 판은 Visual C++ 4.0과 함께 나왔던 MFC 4.0이었는데, 최근 인터넷 기능이 보강된 4.2가 선보였다. MFC에서 제공되는 클래스들은 기존의 윈도우 API를 이용하기 쉽도록 설계되었으며, 대부분의 클래스 멤버함수들은 윈도우 API와 같은 이름을 가지고 있다.

응용프로그램 프레임워크

응용프로그램 마법사(AppWizard)가 만들어 내는 코드를 응용프로그램 프레임 워크라고 부르는데, 이것이 바로 MFC로 만들어진 응용프로그램의 틀(frame)이라고 말할 수 있다.

내용과 보기(Document and View)

하나의 응용프로그램에 대해 그 내부에 사용되는 데이터, 즉 내용(Document) 영역과 실제 화면에 표시하기 위한 코드 부분 즉, 보기(View) 영역을 분리할 수 있도록 내용(Document)과 보기(View)구조가 지원된다.

MFC클래스의 종류

MFC의 클래스들은 크게, CObject 클래스, 응용프로그램 구조 클래스, 시각객체 클래스, 범용 클래스 등으로 나누어 볼 수 있다.

CObject 클래스

대부분의 MFC 클래스들은 바로 이 CObject 클래스에서 계승한 것들이다. 이 CObject클래스는 연속성(serialization)과 실행시 클래스(run-time class) 정보와 디버깅을 위한 진단 출력(diagnostic output)을 지원하게 된다.

연속성(Serialization)이란 객체를 디스크에 저장하거나 디스크에서 읽어오는 것을 말한다.

실행시 클래스 정보를 알아내기 위한 멤버함수로 IsKindOf와 GetRuntimeClass를 제공한다. IsKindOf는 어떤 객체가 특정한 클래스에 속하는지를 알아낸다. GetRuntimeClass는 클래스 이름에 대한 포인터와 CRuntimeClass구조체에 대한 포인터를 얻어준다.

진단(diagnostic) 출력이란 메모리 누설(leak)같은 일이 일어나는 경우에 이 내용을 출력해 준다.

그렇다고 해서 CObject를 상속하는 것이 큰 부담이 되는 것은 아니다. CObject클래스는 단지 네 개의 가상함수와 하나의 CRuntimeClass객체를 가질 뿐이다. 따라서 사용자가 새로 객체를 정의해서 사용하는 경우에 이 CObject 객체를 상속받아 만드는 것이 좋다.

응용프로그램 구조 클래스

윈도 응용프로그램의 시작, 수행, 종료를 모두 책임지는 주 클래스가 바로 CWinApp 클래스이다. 응용프로그램 마법사로 만들어진 하나의 응용프로그램은 CWinApp에서 상속받은 오직 하나의 객체만을 가질 수 있으며, 이 응용프로그램 객체는 창(window)이 만들어지기 전에 만들어 진다.

이 응용프로그램 클래스가 바로 다중 스레드 프로그램에서 주 스레드(primary thread)가 되는 것이다.

MFC가 생성한 코드에는 API 프로그래밍의 WinMain함수가 보이지 않는데, 이 부분은 응용프로그램 객체내에 숨어 있으며, 이 안에서 InitApplication, InitInstance 멤버 함수를 호출하게 된다. 메시지 루프는 Run이란 멤버함수를 호출함으로써 수행된다. 종료할 때는 ExitInstance멤버함수를 호출하게 된다.

...시각객체 클래스

모든 눈에 보이는 객체의 가장 선조 격은 CWnd객체이다. 이 객체는 모든 창 클래스의 기본 객체이다.

보기 클래스(CView)는 내용클래스(CDocument)와 상대되는 개념으로 내용클래스의 내용을 윈도의 클라이언트 영역에 표시해 주는 역할을 한다.

그밖에 대화상자 클래스(CDialog), 제어 클래스(CStatic, CButton, CEdit, CListBox...), 메뉴 클래스(CMenu), 장치 환경(Device Context)와 그리기 객체(CGdiObject, CBitmap, CPen, CFont...)등이 있다.

범용 클래스

CPoint, CSize, CRect, CString, CTime등 일반적으로 쓰이는 클래스들이다.

MFC 프로그래밍

MFC를 사용하기 위해서 꼭 C++에 대해 완벽하게 알아야 하는 것은 아니다. 하지만, 기본적인 C++ 문법 정도는 잘 알아두어야 일단 시작이라도 해볼 수 있으니, 먼저 가물 가물해 진 C++에 대한 아련한 기억을 더듬어 보자.

C++ 기억 더듬기

객체, 클래스, 인스턴스

C++를 처음 접하는 사람은 물론이고, 약간 공부를 한 사람도 객체와 클래스라는 용어를 혼동하는 경우가 많은데, 여기에서 그 의미를 한번 정리해 보자.

객체(object)는 자료와 그 자료를 다루는 함수의 결합체이다. 실세계를 모델링할 때 자료와 그 자료를 다루는 함수를 함께 다루는 것이 편리하다.

이 개체를 프로그래밍에서 구현하는 수단이 되는 것이 바로 클래스(class)이다. C에서 구조체(struct)라는 자료구조가 있는데, 이것에는 여러 가지 변수가 한꺼번에 존재할 수 있는데, 클래스는 이 구조체의 확장형태라고 생각하면 된다. 구조체의 멤버로 변수뿐만 아니라 함수도 가능한 형태가 바로 클래스라고 생각하면 이해가 쉽다. 클래스는 형(type) 개념이다.

인스턴스는 메모리에 확보된 클래스, 즉 변수 개념이다.

```
struct PointStruct {
public:
    int m_x;
    int m_y;
};

class PointClass {
public:
    struct PointStruct m_pos;
    struct PointStruct GetPos();
    void SetPos(struct PointStruct*);
};

PointClass CurPos;
```

Point란 클래스를 만들어 본 것인데, 이 클래스 내에는 m_x와 m_y라는 변수와 m_x, m_y값을 읽어오는 GetPos(). m_x, m_y값을 설정하는 SetPos(struct PointStruct*) 함수가 있다. 클래스가 형(type) 개념이라고 했으니, int나 char처럼 변수를 선언할 수 있다.

PointClass CurPos;

PointClass형의 CurPos라는 변수를 선언한 것이다.

함수 중복정의

함수의 중복정의란 함수의 이름은 같으나 그 반환값의 형이나 인수의 형, 개수 등이 다른 함수를 정의하는 것을 말한다. C로 아주 큰 프로그램을 작성하는 경우에 프로그래머를 곤혹스럽게 하는 것 중 하나가 바로 함수 이름을 짓는 것일 것이다. 모든 함수들은 서로 다른 이름을 가져야 하기 때문이다. 해결책이 하나 있긴 하다. 소스코드를 여러개의 파일로 나누고, 함수들을 각각의 파일 내에서만 사용하게 하는 것이다. 다른 파일에서 사용해야 하는 함수들은 따로 헤더파일에 extern으로 선언해 주고, 나머지 함수들은 모두 함수이름 앞에 static을 붙여주어 해당 파일 내에서만 유효하도록 하는 방법이다.

하지만, C++에서는 같은 이름의 함수가 허용되는데, 그 인수의 개수나 인수의 형이 다르다면, 모두 가능하다는 것이다. 두 변수를 더하는 함수를 예로 들어 보면, 정수 두 개를 더하는 함수와 실수 두 개를 더하는 함수가 필요하다고 하자. C에서는 다음과 같이 AddInt, AddFloat 두 개의 함수이름이 필요할 것이다.

```
AddInt(int a, int b);
AddFloat(float a, float b);
```

하지만, C++에서는 다음과 같이 Add라는 하나의 이름만 있으면 된다. 인수만 다르다면 말이다.

```
Add(int a, int b);
Add(float a, float b);
```

생성자와 소멸자

생성자는 클래스가 생성될 때 컴파일러에 의해 자동으로 수행되며, 사용자가 생성자를 호출할 수는 없다. 생성자이름은 클래스 이름과 같다. 주로, 클래스 멤버변수들의 값 초기화나 필요한 메모리 할당을 위해 사용된다. 인수는 필요한 경우 있을 수 있다.

소멸자는 클래스가 소멸될 때 컴파일러에 의해 자동으로 호출된다. 사용자가 직접 호출할 수는 없다. 소멸자의 이름은 클래스의 이름과 같으나 제일 앞에 ~가 붙는다. 주로, 할당된 메모리 해제 같은 마무리 작업에 사용된다. 소멸자는 생성자와 달리 인수가 없다. 생성자의 경우 함수 중복정의가 가능하나 소멸자의 경우는 클래스 소멸 때 자동으로 수행된다는 특성 때문에 중복정의가 불가능하며 따라서 인수도 있을 수 없다.

생성자나 소멸자를 작성하지 않으면, 컴파일러는 기본적인 생성자와 소멸자를 만들어 생성자나 소멸자가 필요한 경우에 호출하게 된다.

앞에서 다룬 PointClass로 생성자와 소멸자에 대한 예를 만들어 보면 다음과 같다.

```
struct PointStruct {
public:
    int m_x;
    int m_y;
};

class PointClass {
```

```

public:
    PointClass();
    PointClass(struct PointStruct* pos);
    ~PointClass();
    struct PointStruct m_pos;
    struct PointStruct GetPos();
    void SetPos(struct PointStruct*);
};

PointClass::PointClass()
{
    cout << "생성자 호출" << endl;
}

PointClass::PointClass(struct PointStruct* pos)
{
    cout << "인수있는 생성자 호출 " << pos->m_x << ", " << pos->m_y << endl;
}

PointClass::~PointClass()
{
    cout << "소멸자 호출" << endl;
}

struct PointStruct PointClass::GetPos()
{
    return m_pos;
}

void PointClass::SetPos(struct PointStruct* p)
{
    m_pos = *p;
}

main()
{
    PointClass CurPos;                ①
    struct PointStruct Pos;

    CurPos.m_pos.m_x = 5;
    CurPos.m_pos.m_y = 6;

    cout << " x = " << CurPos.m_pos.m_x << " y = " << CurPos.m_pos.m_y << endl;
    Pos.m_x = 10, Pos.m_y = 11;
    CurPos.SetPos(&Pos);

    Pos = CurPos.GetPos();            ②
    cout << " x = " << Pos.m_x << " y = " << Pos.m_y << endl;

    PointClass Pos2(&Pos);            ③, ④
}

```

위의 코드를 컴파일, 링크한 다음 실행시키면 다음과 같은 결과를 얻을 수 있다.

생성자 호출

①

```
x = 5 y = 6
x = 10 y = 11
인수있는 생성자 호출 10,11      ②
소멸자 호출                    ③
소멸자 호출                    ④
Press any key to continue
```

기본 인수(default argument)

기본 인수란 함수의 원형(prototype) 선언에 그 인수를 지정하지 않았을 경우에 자동으로 값이 지정되는 인수를 말한다. 이는 윈도 프로그래밍에서 처럼 인수 개수가 아주 많은 함수들이 많은 경우에 매우 유용하다. 예를 들어 MFC의 CFrameWnd의 Create라는 멤버함수는 다음과 같은 원형을 갖는데, 처음 두 인수를 제외하고는 모두 기본 인수가 설정되어 있음을 볼 수 있다. 특별히 다른 것을 지정하고 싶지 않다면, 처음 두 인수값만 지정해 주어도 된다는 뜻이다.

```
BOOL Create( LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
             DWORD dwStyle = WS_OVERLAPPEDWINDOW,
             const RECT& rect = rectDefault,
             CWnd* pParentWnd = NULL, LPCTSTR lpszMenuName = NULL,
             DWORD dwExStyle = 0, CCreateContext* pContext = NULL );
```

하지만, 예를 들어, pContext같은 인수의 값을 바꾸고 싶다면, 그 앞의 모든 인수들도 다 적어주어야 한다.

포인터형 변수와 참조형 변수

포인터형 변수는 한마디로 주소를 저장할 수 있는 변수이다. Win32에서는 near, far, huge같은 메모리 구분이 없어졌다. 모든 포인터 변수는 far메모리 변수라고 생각해도 좋다. 포인터형 변수의 크기는 Win32에서는 무조건 32비트이다.

참조형 변수는 일반변수같기도 하고, 포인터같기도 하다. 처음 참조형 변수를 접하게 되면 사실 좀 당황하게 되는데, 그 개념이 잘 와닿지 않기 때문일 것이다.

포인터를 배우면서 가장먼저 접하게 되는 예가 바로, 두 변수의 값을 서로 교환하는 예일 것이다.

```

#include <iostream.h>

void SwapValue(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

void SwapPtr(int* x, int* y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

void SwapRef(int& x, int& y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

main()
{
    int x = 5, y = 6;
    SwapValue(x, y);
    cout << "x=" << x << ", y=" << y << endl;
    SwapPtr(&x, &y);
    cout << "x=" << x << ", y=" << y << endl;
    SwapRef(x, y);
    cout << "x=" << x << ", y=" << y << endl;
}

```

함수의 인수 전달 방식에는 값에 의한 전달과 참조에 의한 전달 두가지 방식이 있다. 값에 의한 전달 방식이란, 인수를 전달할 때, 인수는 그 값의 복사본이 전달되기 때문에 전달된 변수의 값의 변화가 그 함수를 호출한 쪽에는 영향을 미치지 못하는 것을 말한다. 참조에 의한 전달이란 인수의 주소를 넘겨주기 때문에 넘겨진 인수의 값을 바꾸면, 그 함수를 호출한 쪽에도 그 값이 전달되는 방식을 말한다. 파스칼에서는 참조에 의한 전달이 지원되기 때문에 위의 예에서 SwapValue()같은 방식으로 함수를 작성하더라도 값이 제대로 바뀌지만, C나 C++에서는 값에 의한 전달만이 지원되기 때문에 SwapValue()와 같이 함수를 작성하면 그 값은 절대로 바뀌지 않는다. 값에 의한 전달 방식에서는 SwapPtr()처럼 그 변수의 주소를 전달해 주고, 함수 내에서는 전달된 포인터형 변수의 내용을 바꿔줌으로써 그 인수의 값을 바꿀 수 있다.

참조형이란 것은 말 그대로 변수를 참조하기 위한 것이다. 참조형 변수를 가장 쉽고 간단하게 이해할 수 있게 정의한다면, 원래의 변수와 같은 주소공간을 갖는 일반변수

라고 정의할 수 있겠다. 이름만 다른 또 하나의 분신! 같은 주소공간을 갖기 때문에 한 변수를 수정하면, 다른 변수도 당연히 그 값이 수정된다. 참조형은 그 특성상 정의와 초기화가 동시에 이루어 져야 한다.

```
int value;  
int& ref;          // Error  
int& ref=value;    // OK
```

세 번째 줄의 ref참조형 변수는 value라는 변수와 같은 주소공간을 갖게된다. 두 번째 줄처럼 참조형 변수만을 정의하는 것은 참조형 변수의 특성상 불가능한 일이다.

앞에서 작성한 예제를 실행시키면 다음과 같은 결과를 얻을 수 있다.

```
x=5, y=6  
x=6, y=5  
x=5, y=6  
Press any key to continue
```

구조체 변수 자체를 인수로 넘겨주는 것은 매우 위험한 일이다. 왜냐하면 인수를 전달할 때, 스택을 이용하는데, 구조체의 경우에는 그 구조체 내부에 아주 큰 멤버변수가 있거나 멤버변수가 아주 많을 경우에는 스택 오버플로가 발생할 수 있기 때문이다. 따라서 인수 전달은 인수에 구조체 변수 자체를 넘겨주지 않고, 보통은 구조체의 포인터를 넘겨주게 된다.

참조형 변수는 C++의 값에 의한 인수 전달 방식의 단점을 극복할 수 있기 때문에 인수의 값을 바꾸는 경우와 구조체의 포인터를 넘겨주는 대신에 사용하는 경우가 많다.

new, delete

메모리를 동적으로 할당하기 위해 malloc이나 free대신에 new와 delete가 사용된다. 10개의 문자형 변수를 동적으로 할당하려면 다음과 같이 해주어야 했다.

C에서는

```
char *pbTemp;
```



```
pbTemp = (char*)malloc(10*sizeof(char));
....
free(pbTemp);
```

C++에서는

```
char *pbTemp;

pbTemp = new char[10];
....
delete[10] pbTemp;
```

new로 생성하고자 하는 것이 위의 예처럼 일반 변수 형이 아니라 클래스인 경우를 생각해 보면, new를 통해 클래스가 생성될 때 생성자가 호출된다. 물론 delete를 통해서도 소멸자가 호출된다.

인라인 함수

C++의 함수에는 C의 매크로와 비슷한 개념의 함수가 존재한다. C에서는 다음과 같은 매크로를 이용해 절대값을 구하는 함수를 대신할 수 있었다. 매크로를 이용하면, 매크로는 선행처리되어 소스코드 내에 대치되기 때문에 함수보다 훨씬 수행이 빠르다. 때문에 자주 사용되는 간단한 함수같은 경우에는 매크로로 정의해서 사용하는 것이 좋은 경우도 많다. 하지만 매크로로 작성된 함수의 경우에 결정적인 약점이 있는데, 그것은 바로 형(type)을 검사할 수 없다는 것이다.

```
#define abs(a) ((a) > 0) ? (a) : (-a)
```

매크로 함수의 장점만을 따서 C++에서 사용하는 함수가 바로 인라인 함수이다. 인라인 함수는 일반 함수처럼 형 검사도 수행되고, 자동 형변환도 수행된다. 또 매크로 함수처럼, 함수가 적절히 전개되어 컴파일된다. 인라인 함수는 함수 앞에 inline이라고 밝혀 주거나 또는 클래스 내부에 함수 본체를 직접 작성해 주면 자동으로 인라인 함수로 등록된다. 앞에서 들었던 PointClass의 예를 인라인 함수 형태로 수정해보자.

```
#include <iostream.h>

struct PointStruct {
```

```

public:
    int m_x;
    int m_y;
};

class PointClass {
public:
    struct PointStruct m_pos;

    struct PointStruct GetPos() { return m_pos; }
    void SetPos(struct PointStruct* p) { m_pos = *p; }
};

main()
{
    PointClass CurPos;
    struct PointStruct Pos;

    CurPos.m_pos.m_x = 5;
    CurPos.m_pos.m_y = 6;

    cout << " x = " << CurPos.m_pos.m_x << " y = " << CurPos.m_pos.m_y << endl;
    Pos.m_x = 10, Pos.m_y = 11;
    CurPos.SetPos(&Pos);

    Pos = CurPos.GetPos();
    cout << " x = " << Pos.m_x << " y = " << Pos.m_y << endl;
}

```

프로그램이 훨씬 간단해 보인다는 것을 알 수 있을 것이다. 그렇다고 인라인 함수를 아무때나 사용해서도 안되고, 사용할 수도 없다. 인라인 함수는 말그대로 inline, 몇줄 정도로 충분한 코드인 경우에만 사용하도록 한다. 기존의 매크로 함수를 사용하던 곳에 사용하면 아주 적절하다. 인라인 함수의 제한점을 보면

루프문(while, for, do while)이나 switch, goto문을 사용할 수 없다.

재귀호출을 할 수 없다.

인라인 함수의 주소를 함수 포인터로 얻어 사용할 수 없다. 왜냐하면 인라인 함수는 호출되는 것이 아니라 해당 부분에 치환되어 사용되기 때문이다.

컴파일러마다 차이가 있긴 하겠지만, 부적절한 문을 사용하면 인라인 함수는 자동으로 일반함수로 컴파일이 되므로, 적절하게 작성해서 사용해서 인라인 함수만의 잇점을 얻을 수가 있다.

상속

C++의 가장 큰 특징 중 하나이다. 캡슐화같은 특성의 경우에는 잘 작성된 C프로그램으로도 충분히 가능한 일이지만, 상속이야말로 C++만의 특성이자 장점이다. 상속을 통해 상위클래스의 성질을 그대로 이어받고, 자신의 특성을 추가해서 새로운 클래스를 만들어 사용함으로써, 프로그램의 재사용성을 높여준다. 우리가 MFC로 손쉽게 윈도우 프로그램을 작성할 수 있는 것도, 바로 MFC에 수많은 클래스들이 정의되어 있고, 우리는 이 클래스들 중에 원하는 용도에 알맞은 클래스를 상속받아 사용함으로써 가능한 일인 것이다.

Point라는 클래스와 이를 상속받아 만든 Circle이란 클래스를 보자. Point에는 위치 정보인 m_x, m_y와 색 정보인 m_color와 이 멤버 변수를 다루기 위한 몇가지 멤버함수가 인라인 함수로 정의되어 있다.

Circle클래스는 Point클래스를 public으로 상속받고, Circle자체는 반지름을 나타내는 m_radius멤버변수만을 갖는다.

```
#include <iostream.h>

class Point {
protected:
    int m_x, m_y;
    int m_color;
public:
    int  GetPosX() { return m_x; }
    int  GetPosY() { return m_y; }
    int  GetColor() { return m_color; }
    void SetPos(int x, int y) { m_x = x, m_y = y; }
    void SetColor(int color) { m_color = color; }
};

class Circle : public Point {
protected:
    int m_radius;
public:
    int  GetRadius() { return m_radius; }
    void SetRadius(int radius) { m_radius = radius; }
};

main()
{
    Circle myCir;

    myCir.SetPos(5, 6);
    myCir.SetColor(255);
    myCir.SetRadius(10);

    cout << "x=" << myCir.GetPosX() << endl;
    cout << "y=" << myCir.GetPosY() << endl;
    cout << "color=" << myCir.GetColor() << endl;
    cout << "radius=" << myCir.GetRadius() << endl;
}
```

하지만, 위의 예에서 보듯이 상속받은 부모 클래스의 멤버변수나 함수를 마치 자신의 멤버변수나 함수처럼 사용할 수 있다. 즉, 클래스를 상속받을 때, 상속받은 클래스에는 상위 클래스의 멤버변수에 해당되는 메모리 공간이 상속받은 클래스 인스턴스내에도 확보된다. 상속의 개념을 이해하는데 가장 중요한 것은 바로, 상위 클래스 멤버변수가 상속받는 클래스의 인스턴스 내에도 할당된다는 바로 그 점이다. 만약 Circle클래스를 상속받은 Ellipse라는 클래스가 있다면, 이 클래스의 인스턴스에는 Point와 Circle의 멤버변수를 위한 메모리 공간이 할당된다는 것이다. 위의 예를 컴파일, 링크해서 실행시키면 다음과 같은 결과를 얻을 수 있다.

```
x=5
y=6
color=255
radius=10
Press any key to continue
```

함수의 중복정의(overloading)와 함수의 재정의(overriding)

함수의 중복정의란 앞서에서도 살펴보았듯이 함수의 이름은 같지만 함수의 인수나 반환형이 다른 함수를 만드는 것을 말한다. 그렇다면 함수의 재정의란 무엇일까? 함수의 재정의란, 상속관계에 있는 클래스에서 부모 클래스에 있는 함수와 완전히 똑같은 함수를 말한다. 인수와 인수 개수, 반환형까지 말이다.

```
#include <iostream.h>

class Point {
protected:
    int m_x, m_y;
    int m_color;
public:
    int GetPosX() { return m_x; }
    int GetPosY() { return m_y; }
    int GetColor()
    {
        cout << "Point get color" << endl;
        return m_color;
    }
    void SetPos(int x, int y) { m_x = x, m_y = y; }
    void SetColor(int color) { m_color = color; }
};

class Circle : public Point {
```

```

protected:
    int m_radius;
public:
    int GetColor()
    {
        Point::GetColor();
        cout << "Circle get color" << endl;
        return -1;
    }
    int GetRadius() { return m_radius; }
    void SetRadius(int radius) { m_radius = radius; }
};

main()
{
    Circle myCir;
    myCir.SetPos(5, 6);
    myCir.SetColor(255);
    myCir.SetRadius(10);
    cout << "x=" << myCir.GetPosX() << endl;
    cout << "y=" << myCir.GetPosY() << endl;
    cout << "color=" << myCir.GetColor() << endl;
    cout << "radius=" << myCir.GetRadius() << endl;
}

```

이 코드를 컴파일, 링크한 다음 실행시키면 다음과 같은 결과를 얻을 수 있다.

```

x=5
y=6
Point get color
Circle get color
color=-1
radius=10
Press any key to continue

```

함수 재정의가 도대체 어디에 쓰일까? 자신이 다른 사람이 사용하게 될 클래스를 작성한다고 하자. 그런데, 그 클래스의 특정한 함수는 꼭 불러서 TRUE를 반환해야만 한다고 하자. 이런 경우 하위 클래스에서는 상위 클래스의 멤버함수와 똑같은 멤버함수를 작성해서 조건에 맞는 경우, TRUE를 반환하도록 작성한다. 이 함수 재정의는 응용 프로그램 마법사(AppWizard)가 만들어내는 MFC의 응용프로그램 틀(framework)에서 아주 유용하게 사용된다.

처음 만들어 보는 MFC 프로그래

밍

안녕? 윈도우!

지금까지 배운 것을 기반으로 이제, 내 손으로 직접 MFC 프로그래밍에 도전해 보자.
흔히 하듯이 화면 가운데 "안녕? 윈도우!"를 출력하는 프로그램을 작성해 보자.

시작하기에 앞서 CWinApp에 대해 우선 중요한 것들부터라도 살펴보고 넘어가자.

CWinApp의 멤버 변수

변수이름	설명
HINSTANCE m_hInstance	현재 응용프로그램의 인스턴스 핸들. WinMain에 인자로 전달되는 hInstance와 같은 것으로 생각하면 된다. AfxGetInstanceHandle()로 얻을 수 있다.
HINSTANCE m_hPrevInstance	응용프로그램의 이전 인스턴스 핸들. WinMain에 인자로 전달되는 hInstance와 같은 것으로 생각하면 되는데, Win32환경에서는 각 응용프로그램이 모두 독자적인 메모리 공간을 갖기 때문에 항상 NULL이다.
LPSTR m_lpCmdLine	명령행 문자열에 대한 포인터. WinMain의 lpCmdLine에 해당한다.
int m_nCmdShow	프로그램의 창 표시 형식을 지정한다. ShowWindow의 인자로 사용되며 WinMain의 nCmdShow에 해당한다.
CWnd* m_pMainWnd	주 창에 대한 CWnd클래스 포인터
const char* m_pszAppName	CWinApp의 생성자에서 지정해 준 실행파일의 이름. 여기에서 지정해 주지 않았다면 문자열 자원 (r e s o u r c e) 의 AFX_IDS_APP_TITLE에서 지정해 준 값을 갖는다. 이것도 지정해 주지 않았다면 .EXE 실행파일의 이름으로 설정된다.
const char* m_pszExeName	응용프로그램의 실행파일의 이름 문자열에 대한 포인터. 확장자 .EXE는 포함되지 않으며, m_pszAppName과는 달리 공백은 포함될 수 없다.
const char* m_pszHelpFilePath	도움말 파일이 있는 곳의 경로와 파일 이름 문자열에 대한 포인터
const char* m_pszProfileName	INI파일 이름 문자열에 대한 포인터
LPCTSTR m_pszRegistryKey	시스템 레지스트리에 저장할 때 사용되는 키 이름 문자열에 대한 포인터. SetRegistryKey함수를 이용해서 설정한다.

먼저 HELLO.H란 이름으로 파일을 하나 새로 만들어서 응용프로그램 구조 개체인 CWinApp를 상속해서 CHelloApp란 클래스를 만들자. 이 클래스의 멤버 함수로는 InitInstance()를 둔다. InitInstance함수는 CWinApp의 오버라이딩 함수로서 함수이름과 함수 인자 그 반환값 형식까지 완전히 똑같이 사용해야 한다.

InitInstance에서는 프로그램이 시작될 때 필요한 여러 가지 초기화 작업을 해주면 좋다. 이 함수가 FALSE를 돌려주면, 프로그램은 더 이상 계속되지 못하고 그냥 종료된다.

CWinApp에는 여러개의 오버라이드 가능한 함수들이 있지만, 다른 것들은 대개 잘 사용되지 않지만 InitInstance는 거의 대부분 오버라이드해서 사용한다.

```
class CHelloApp : public CWinApp
{
public:
    BOOL InitInstance();
};
```

그 다음으로, 거의 대부분의 윈도 응용프로그램은 하나 이상의 창을 갖게 되는데, 일반적인 윈도 응용프로그램에서 보는 모양의 창을 만들려면 CFrameWnd에서 상속 받아 클래스를 하나 만들어야 한다. 이 클래스에는 그냥 생성자 하나만을 만들기로 한다. 이 생성자에서 주 창을 만들기로 하자.

```
class CHelloWnd : public CFrameWnd
{
public:
    CHelloWnd();
};
```

이것으로 일단, HELLO.H의 기본적인 준비는 마친 셈이다. 이제 HELLO.CPP를 작성해 보자. 먼저 CWinApp를 상속받아 만든 CHelloApp클래스의 객체를 하나 만든다. 이 객체는 오직 하나만 존재 할 수 있다.

```
CHelloApp helloApp;
```

그 다음, CHelloApp의 InitInstance함수를 작성한다. 여기에서는 주 창을 만들어 화

면에 나타내는 것까지 처리한다.

```
BOOL CHelloApp::InitInstance()
{
    m_pMainWnd = new CHelloWnd();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
```

CWinApp의 멤버 변수인 m_pMainWnd에는 새로 만드는 주 창에 대한 포인터가 할당되어야 하는데, 주 창은 CFrameWnd의 멤버함수인 Create로 만들 수 있다. CFrameWnd를 상속받아 만든 CHelloWnd가 있으니 CHelloWnd의 생성자에서 주 창을 만들기로 하고 일단 다음 줄로 넘어간다. ShowWindow와 UpdateWindow는 익숙한 함수이다. 하지만 이들 함수는 소스 코드에서 보듯이 CFrameWnd의 멤버함수이다. MFC는 이처럼 윈도우 API와 똑같은 함수이름을 많이 사용함으로써 기존 윈도우 프로그래머들이 새로 MFC의 함수들을 외울 필요를 줄여준다.

이제 CHelloWnd의 생성자에 대한 코드를 보자.

```
CHelloWnd::CHelloWnd()
{
    Create(NULL, "안녕");
}
```

아주 간단하게 Create함수만이 있는데, 이 함수는 CreateWindow와 RegisterWindow라는 API에 대응되는 함수이다.

```
BOOL Create( LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
             DWORD dwStyle = WS_OVERLAPPEDWINDOW,
             const RECT& rect = rectDefault,
             CWnd* pParentWnd = NULL, LPCTSTR lpszMenuName = NULL,
             DWORD dwExStyle = 0, CCreateContext* pContext = NULL );
```

원래는 이와같이 많은 인자가 있지만 처음 두 개만 빼고는 모두 기본 인자가 있기 때문에 일단 나머지는 설정하지 않았다.

이제 화면에 "안녕? 윈도우!"라는 문자열을 표시해 주는 부분만 작성해 주면되는데, 지금까지 작성한 소스코드를 생각해 보면 한가지 의문점이 자연 생겨날 것이다. 메시지가 루프가 보이지 않는다는 것과, 과연 이 프로그램이 어디에서 시작하고 있는지 알

수가 없다는 것이다. WinMain에 해당하는 함수가 보이지 않기 때문인데, 한마디로 말해 모든 것은 바로 CWinApp 안에 있다. MFC로 작성된 응용프로그램을 디버거로 처음부터 따라가다보면, WinMain비슷한 것과 메시지 루프 비슷한 것들을 만날 수 있다.

MFC에서는 메시지 루프 대신, CCmdTarget클래스를 상속받아 사용하면, 메시지를 다룰 수 있게 된다. CWnd, CWinThread, CDocument, CView등의 클래스들도 모두 CCmdTartget클래스를 상위 클래스로 갖는다. 그리고, 클래스 끝부분에 다음과 같은 매크로를 써주면, 그 클래스가 메시지를 다룰 수 있게 된다.

```
DECLARE_MESSAGE_MAP();
```

SDK에서는 메시지 루프와 윈도 프로시저가 하나의 쌍으로서 메시지를 처리해 주었는데, MFC에서는 메시지 루프 대신, CCmdTarget을 상속받고, 윈도 프로시저 대신, 메시지 처리기(Handler)함수가 사용된다. WM_XXX메시지에 해당하는 메시지 처리함수는 OnXXX()식의 이름을 갖는다. 예를 들어, WM_PAINT라는 메시지에 대응하는 메시지 처리함수는 OnPaint()이다. SDK 프로그래밍에서는 윈도 프로시저에서 각 메시지에 대한 case문으로 메시지를 처리해 주었는데 이는 사실 한 함수가 너무 길어지게 되는 등 별로 바람직한 코딩 방식은 아니었다. 이에 비해 MFC의 메시지 처리 방식은 CCmdTaget클래스를 상속받은 클래스의 멤버함수로서 OnXXX()형식으로 구현되므로, 코딩이 아주 산뜻해 졌다고 할 수 있겠다. 이제 완성된 형태의 HELLO.H를 만들어 보자.

<HELLO.H>

```
class CHelloApp : public CWinApp
{
public:
    BOOL InitInstance();
};

class CHelloWnd : public CFrameWnd
{
public:
    CHelloWnd();

protected:
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP();
};
```

afx_msg는 메시지 처리 함수 앞에 관례적으로 붙이는 것으로 사실 아무 의미없는 것이다. 여기에서 한가지 의문이 더 생길 수 있다. OnPaint()같은 함수의 경우에 void

형에 인수는 없는데, 그렇다면, 각 메시지 처리 함수들을 그 때 그 때마다 도움말을 찾아보면서 작성해야 하는가 하는 점이다. 만일 그렇다면 아주 번거로운 일이 될 것이다. 물론 위와 같은 식으로 코드를 직접 작성하는 경우에는 그렇게 하는 수밖에 다른 방법은 없을 것이다. 하지만, 응용프로그램 마법사(AppWizard)로 만들어내는 응용프로그램 틀(Application Framework)을 기반으로, 클래스 마법사(ClassWizard)를 이용해서 메시지 처리함수를 작성할 때는 함수의 반환형이나 함수의 인수등이 자동으로 생성되기 때문에 걱정할 필요가 없다. 응용프로그램 마법사가 만들어내는 응용프로그램 틀을 이해한다면, 훨씬 쉽고 간편하게 응용프로그램을 작성할 수 있다.

메시지 처리함수는 메시지 처리 지도를 이용해서 정의한다. 메시지 처리함수는

BEGIN_MESSAGE_MAP(..)과 END_MESSAGE_MAP()이란 매크로 사이에 ON_WM_XXX형식으로 기술된다. 처리되지 않은 메시지는 이 클래스의 상위 클래스로 전달되는데, BEGIN_MESSAGE_MAP 매크로에는 메시지 처리기가 정의된 클래스의 이름과, 그 클래스의 상위 클래스의 이름이 명시되는데, 이는 이 클래스에서 처리되지 않은 메시지를 전달할 상위 클래스를 알기 위해서이다. MFC의 메시지 처리 방식이 SDK에서 사용되던 메시지 루프와 윈도 프로시저와 다르기 때문에 MFC에서 메시지가 어떻게 흘러가는지 이해하는 것은 매우 중요하다.

```
BEGIN_MESSAGE_MAP(CHelloWnd, CFrameWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

<HELLO.CPP>

```
#include <afxwin.h>
#include "hello.h"

CHelloApp helloApp;

BOOL CHelloApp::InitInstance()
{
    m_pMainWnd = new CHelloWnd();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

BEGIN_MESSAGE_MAP(CHelloWnd, CFrameWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()

CHelloWnd::CHelloWnd()
{
    Create(NULL, "안녕");
}
```

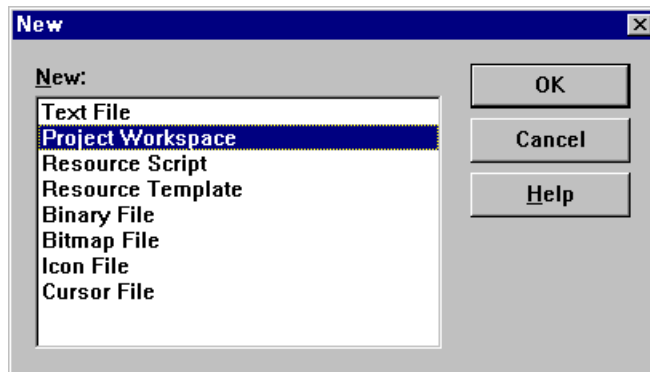
```

void CHelloWnd::OnPaint()
{
    CRect rect;
    CPaintDC dc(this);

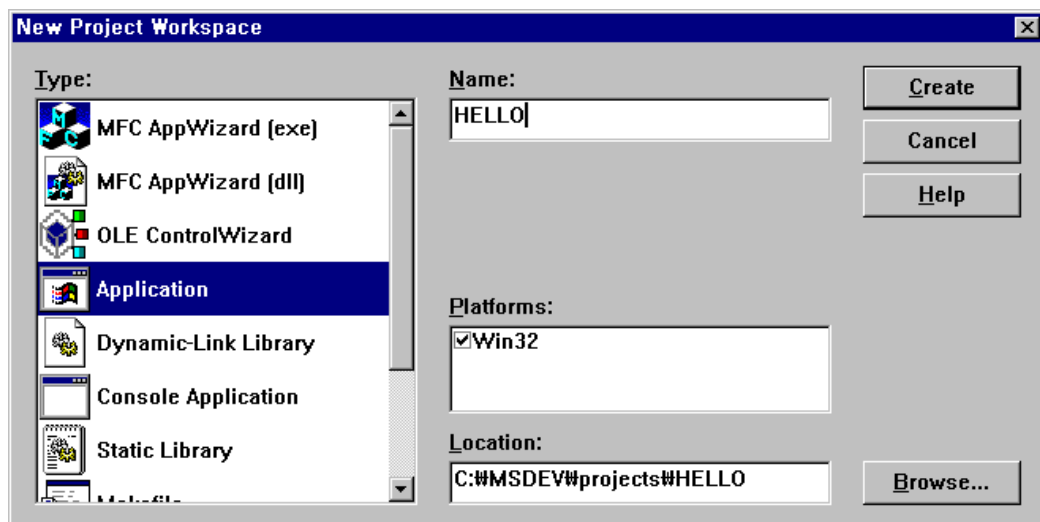
    GetClientRect(rect);
    dc.DrawText("안녕? 윈도우!", -1, rect, DT_SINGLELINE|DT_CENTER|DT_VCENTER);
}

```

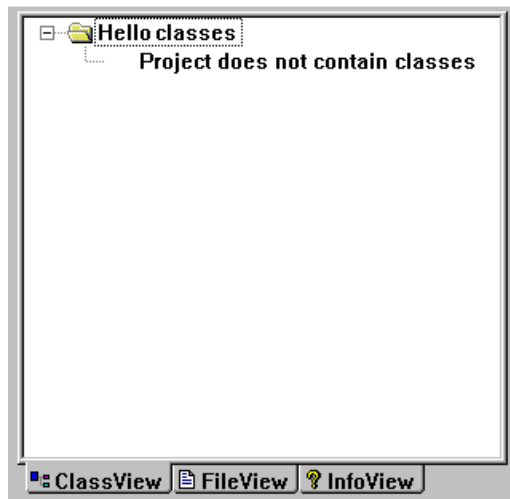
자, 이렇게 해서 HELLO.H와 HELLO.CPP를 작성했는데, 컴파일과 링크를 해서 실행시켜 보자. 컴파일을 하려면, 먼저 HELLO.MAK파일을 만들어야 하는데, 물론 직접 작성할 필요는 없다. 파일 메뉴에서 New를 선택하고, Project workspace를 고른다.



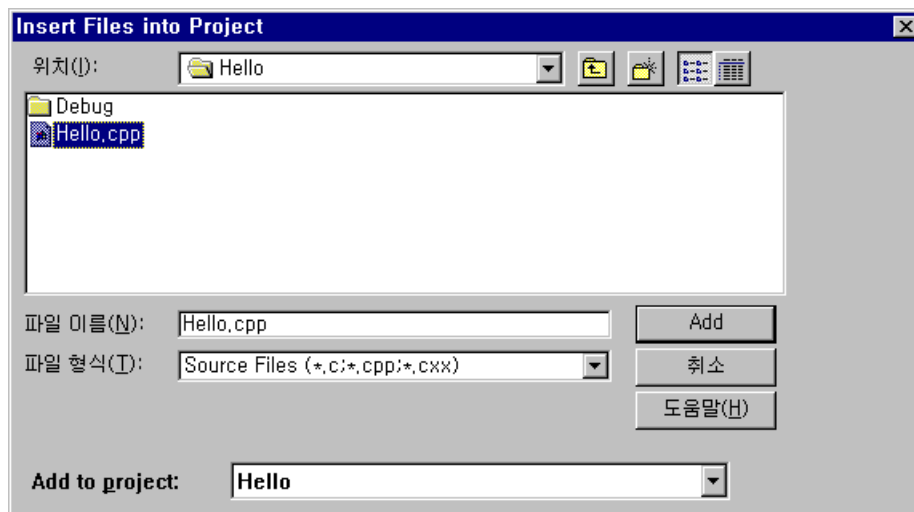
New Project Workspace라는 대화상자가 나타나는데, 지금은 AppWizard를 사용하는 것이 아니므로, 여기에서는 Application을 선택한다. Name:항목에는 HELLO라고 쓰자. 그렇게 하면, Location:항목에 C:\MSDEV\projects\HELLO라고 'HELLO'가 자동으로 추가되는 것을 볼 수 있다. HELLO라는 디렉토리가 따로 생성될 것이다.



여기에서 **Create** 버튼을 누르면 응용프로그램 마법사와는 달리 아무일 없이 그냥 끝날 것이다. 단지 프로젝트 워크스페이스에 다음과 같이 "Hello classes"라는 것만 나타날 것이다. 여기까지만 하면, HELLO.MAK파일은 만들어진 것이다. 이제 여기에 우리가 작성한 HELLO.H와 HELLO.CPP를 추가하면 된다. "Hello classes"를 마우스로 눌러보면 Project does not contain classes라는 메시지만이 표시될 것이다.

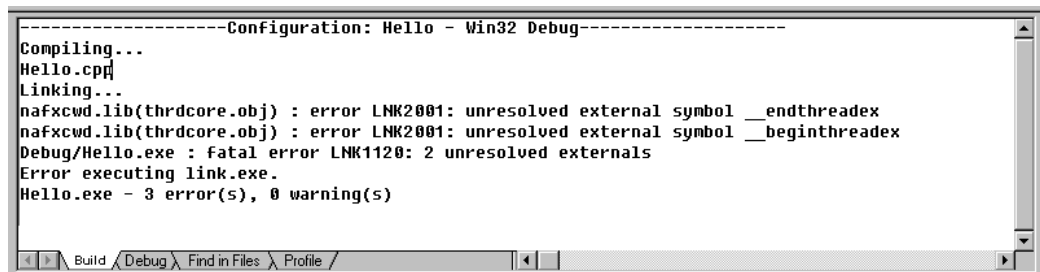


여기에 우리가 작성한 파일을 추가하는 방법은 다음과 같다. Insert메뉴에서 Files into Project메뉴를 선택한 다음, HELLO.CPP를 선택하면 된다. HELLO.H는 따로 선택하지 않아도 되는데, 이것은 HELLO.CPP에서 #include "hello.h"를 해주었기 때문이다. include된 사용자 헤더 파일은 CPP파일을 프로젝트에 추가하면 자동으로 추가되게 된다.

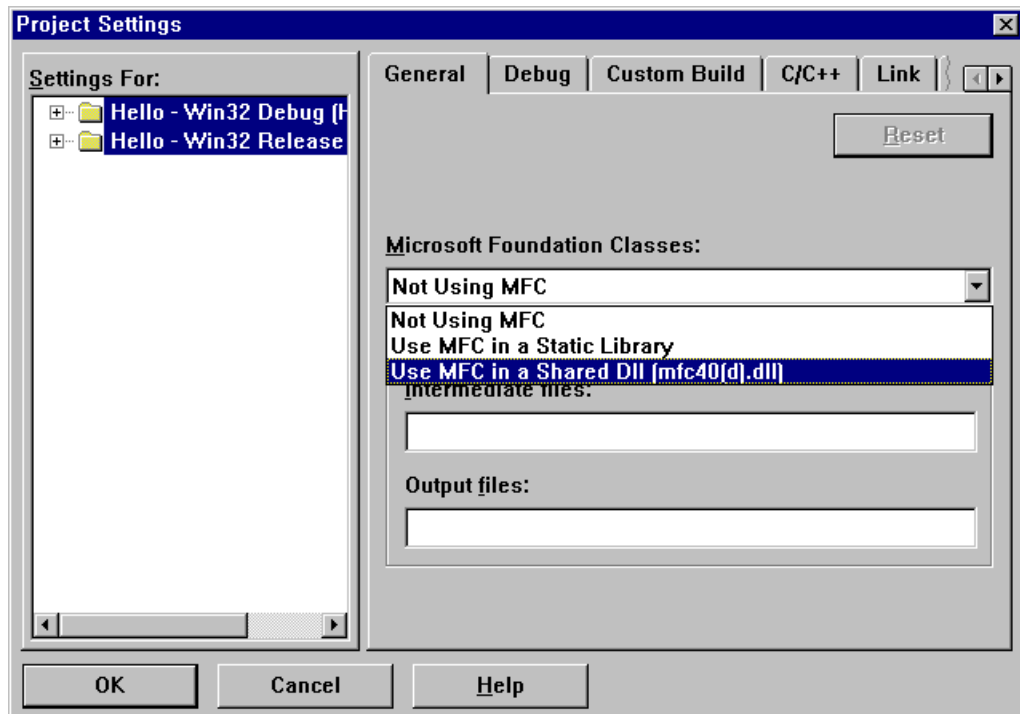


자, 이것으로서 모든 준비는 마친 것 같다. Build메뉴의 Build hello.exe를 선택해서 컴파일과 링크를 해보자. 그런데, 이게 웬일? 컴파일은 무사히 끝났는데, 링크에서 다음과 같은 오류가 발생했다. 왜 일까? 문제는 make파일에 있었다. 프로젝트 워크스페이스

를 선택할 때 그냥 Application을 선택했기 때문에 그냥, 일반 SDK프로그램용으로 make파일이 작성되었기 때문이다. 이 문제를 해결하려면 Build의 Settings를 바꿔주면 된다.



프로젝트 설정 대화상자에서 Not Using MFC를 Use MFC in a Shared DLL로 선택해 주어야 MFC용으로 작성된 프로그램을 컴파일, 링크 할 수 있게 된다.



자, 이제 진짜 모든 준비가 끝났으니, Build 메뉴의 Build hello.exe를 해 보자. 물론 아무 오류없이 잘 끝날 것이다. Build 메뉴의 Execute hello.exe를 선택해서 프로그램을 실행시켜보자. 다음 그림과 같은 응용프로그램이 나타날 것이다. 윈도의 크기를 조절하더라도 "안녕? 윈도야!"라는 말은 언제나 클라이언트 영역의 가운데 나타날 것이다.

