Microsoft's Foundation Class

Overview:

Just to scare you, do the following. Go to Search and search under Index for:

CFrameWnd

Choose CFrameWnd from the titles. Go to the bottom of this page to:

Select Hierarchy Chart.

Microsoft's blurb:

> The Microsoft Foundation Class Library (MFC) is an "application framework" for programming in Microsoft Windows. Written in C++, MFC provides much of the code necessary for managing windows, menus, and dialog boxes; performing basic input/output; storing collections of data objects; and so on. All you need to do is add your application-specific code into this framework. And, given the nature of C++ class programming, it's easy to extend or override the basic functionality the MFC framework supplies.
>
> The MFC framework is a powerful approach that lets you build upon the work of expert programmers for Windows. MFC shortens development time; makes code more portable; provides tremendous support without reducing programming freedom and flexibility; and gives easy access to "hard to program" user-interface elements and technologies, like ActiveX, OLE, and Internet programming. Furthermore, MFC simplifies database programming through Data Access Objects (DAO) and Open Database Connectivity (ODBC), and network programming through Windows Sockets. MFC makes it easy to program features like property sheets ("tab dialogs"), print preview, and floating, customizable toolbars.

MFC is loosely organized into several major categories:

Window Support classes

Application Architecture classes

Graphics Support classes

System Support classes

Collection classes

Non-CObject classes

Most of the MFC classes have a CObject base class.

Base class: CObject

Construction

Cobject	Default constructor.
Cobject	Copy constructor.
operator new	Special new operator.
operator delete	Special delete operator.
operator =	Assignment operator.

Diagnostics

AssertValid	Validates this object's integrity.
Dump	Produces a diagnostic dump of this object.

Serialization

IsSerializable	Tests to see whether this object can be serialized.
Serialize	Loads or stores an object from/to an archive.

Miscellaneous

GetRuntimeClass	Returns the CRuntimeClass structure corresponding to this object's class.
IsKindOf	Tests this object's relationship to a given class.

Serialization:

Serialization is the conversion of an object to and from a persistent form. That is, it means writing or reading an object from any persistent storage (like a disk, clipboard, or for OLE embedding).

MFC uses a CArchive object for serialization. These represent some persistent storage.

The idea is that each object knows best how to store itself. It knows its internal structures and can deal with them. CArchive objects know how to transfer the resulting data stream to persistent media.

For example:

```
#include <afxwin.h>
#include <afxext.h>

class String : public CObject
{
DECLARE_SERIAL(String)
public:
    String( const char * pszIn = "" );
    ~String()       { delete [] m_pszData; };
    virtual void Serialize( CArchive &ar);
    char    *m_pszData;
    WORD        m_nSize;                // WORD is always 16 bits
};

// in the implementation file
IMPLEMENT_SERIAL(String, CObject, 0)

void String::Serialize(CArchive &ar)
{
    if ( ar.IsStoring() ) {
            ar << m_nSize;
            ar.Write(m_pszData, m_nSize);
    }
    else {
            ar >> m_nSize;
            m_pszData = new char[m_nSize];
            ar.Read( m_pszData, m_nSize);
    }
}
```

```
String::String( const char * pszIn )
{
    m_pszData = new char[m_nSize = strlen( pszIn) + 1 ];
    strcpy( m_pszData, pszIn );
}
```

What is the DECLARE/IMPLEMENT stuff all about???

Runtime Type Information:

There is a class in MFC called CRunTimeClass that has member variables that hold names of other classes and the size of their objects. This is used to identify objects and is used in serialization. Macros are used to implement the CRunTimeClass. Here are three important ones:

DECLARE_DYNAMIC	Adds runtime info to the class
IMPLEMENT_DYNAMIC	Enables use of IsKindOf
DECLARE_DYNCREATE	Makes class dynamically creatable
IMPLEMENT_DYNCREATE	through CRunTimeClass::CreateObject
DECLARE_SERIAL	Adds serialization capability
IMPLEMENT_SERIAL	Enables << and >> for CArchive

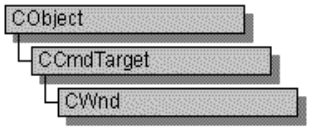
The declare goes in the class declaration and the implement in the implementation file. Only one set is needed.

Windows and MFC Object Distinction:

MFC classes **represent** various types of objects like windows, device contexts (a device context is an interface to a specific graphic device), or GDI objects. Windows 95/NT objects are distinct. We might have a Windows object but no corresponding MFC object. Conversely, we might have an MFC object without a corresponding Windows object. For example, a CWnd object just represents a window. It might get attached to a Windows window.

Initial Classes:

Here are the first three classes of interest:



CCmdTarget:

CCmdTarget is the base class for the Microsoft Foundation Class Library message-map architecture. A message map routes commands or messages to the member functions you write to handle them. (A command is a message from a menu item, command button, or accelerator key.

CWnd

The CWnd class provides the base functionality of all window classes in the Microsoft Foundation Class Library.

A CWnd object is distinct from a Windows window, but the two are tightly linked. A CWnd object is created or destroyed by the CWnd constructor and destructor. The Windows window, on the other hand, is a data structure internal to Windows that is created by a Create member function and destroyed by the CWnd virtual destructor. The DestroyWindow function destroys the Windows window without destroying the object.

Within the Microsoft Foundation Class Library, further classes are derived from CWnd to provide specific window types. Many of these classes, including CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView, and CDialog, are designed for further derivation. The control classes derived from CWnd, such as CButton, can be used directly or can be used for further derivation of classes.

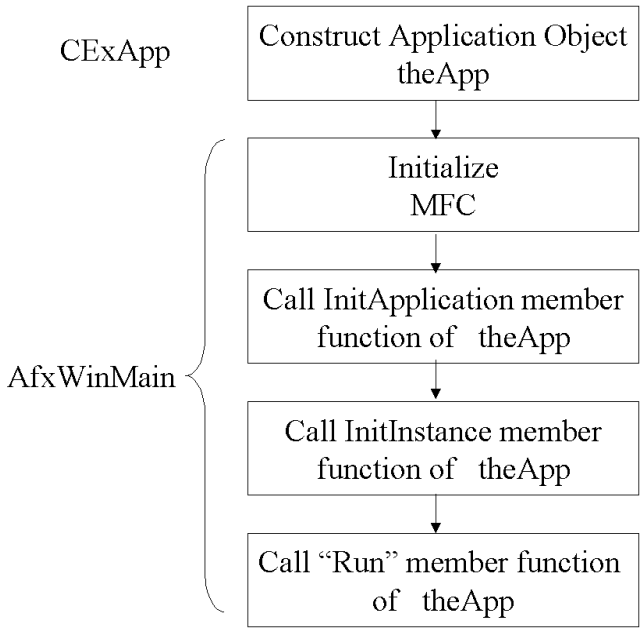
#include <afxwin.h>

An Example Program:

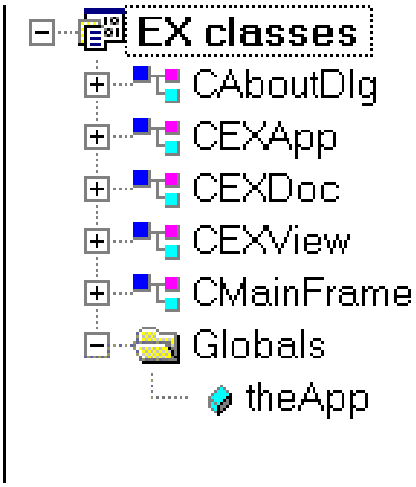
Do the following:

New
MFC AppWizard (EXE)
Name it EX
Step 1: Choose Single Document Interface
Step 4: Select "Advanced" button
Enter GJK as the File Extension
Enter Hello Folks as the main frame caption

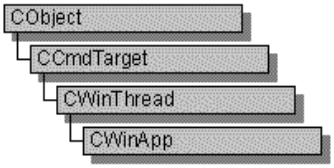
Here's the sequence of what happens when you run an MFC AppWizard program:



After the project is created look at the class view. You should have:



theApp comes from CEXApp and CEXApp is derived from CWinApp.



The CWinApp class is the base class from which you derive a Windows application object. An application object provides member functions for initializing your application (and each instance of it) and for running the application.

Each application that uses the Microsoft Foundation classes can only contain **one** object derived from CWinApp. This object is constructed when other C++ global objects are constructed and is already available when Windows calls the WinMain function, which is supplied by the Microsoft Foundation Class Library. Declare your derived CWinApp object at the global level.

Here are Overrideable member function:

InitInstance	override to perform Windows instance initialization, such as creating your window objects.
Run	Runs the default message loop. Override to customize the message loop.

When you derive an application class from CWinApp, override the **InitInstance** member function to create your application's main window object.

InitApplication can also be overridden but this is for each new instance which is rare.

Here's the CEXApp class

```
class CEXApp : public CWinApp
{
public:
	CEXApp();

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CEXApp)
	public:
	virtual BOOL InitInstance();
	//}}AFX_VIRTUAL

// Implementation

	//{{AFX_MSG(CEXApp)
	afx_msg void OnAppAbout();
	// NOTE - the ClassWizard will add and remove member functions here.
	//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Here's the implementation.

```
// EX.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "EX.h"

#include "MainFrm.h"
#include "EXDoc.h"
#include "EXView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
```

```
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CEXApp
```

We'll talk about message maps below.

```
BEGIN_MESSAGE_MAP(CEXApp, CWinApp)
        //{{AFX_MSG_MAP(CEXApp)
        ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG_MAP
        // Standard file based document commands
        ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
        ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
        // Standard print setup command
        ON_COMMAND(ID_FILE_PRINT_SETUP,
                CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CEXApp construction

CEXApp::CEXApp()
{
        // TODO: add construction code here,
        // Place all significant initialization in InitInstance
}
```

Here is the application object:

```
/////////////////////////////////////////////////////////////////////////////
// The one and only CEXApp object

CEXApp theApp;
```

Here are the inititializations corresponding to the choices we made in AppWizard.

```
/////////////////////////////////////////////////////////////////////////////
// CEXApp initialization

BOOL CEXApp::InitInstance()
{
```

Note: any function beginning with Afx is a global MFC function - not part of a class.

```
        AfxEnableControlContainer();

        // Standard initialization
        // If you are not using these features and wish to reduce the size
        //  of your final executable, you should remove from the following
        //  the specific initialization routines you do not need.

#ifdef _AFXDLL
        Enable3dControls();   // Call this when using MFC in a shared DLL
#else
        Enable3dControlsStatic();      // Call this when linking to MFC statically
#endif

        // Change the registry key under which our settings are stored.
        // You should modify this string to be something appropriate
        // such as the name of your company or organization.
        SetRegistryKey(_T("Local AppWizard-Generated Applications"));

        // Load standard INI file options (including MRU)
        LoadStdProfileSettings();

        // Register the application's document templates.  Document templates
        //  serve as the connection between documents, frame windows and views.
```

Here is where the document template is created.

```
        CSingleDocTemplate* pDocTemplate;
        pDocTemplate = new CSingleDocTemplate(
                IDR_MAINFRAME,
                RUNTIME_CLASS(CEXDoc),
                RUNTIME_CLASS(CMainFrame),      // main SDI frame window
                RUNTIME_CLASS(CEXView));
```

And added to the application's document templates. This is used when a user selects File:New. When CWinApp::OnFileNew is run, it creates this document.

```
        AddDocTemplate(pDocTemplate);

        // Enable DDE Execute open
        EnableShellOpen();
        RegisterShellFileTypes(TRUE);
```

```
	// Parse command line for standard shell commands, DDE, file open
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;

	// The one and only window has been initialized, so show and update it.
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	// Enable drag/drop open
	m_pMainWnd->DragAcceptFiles();

	return TRUE;
}
```

The rest contains the CAboutDlg class details. We'll come back to that.

Message Maps:

The Run member function on CWinApp dispatches messages to the target windows like the non-MFC used a message loop. It even calls ::DispatchMessage too!!

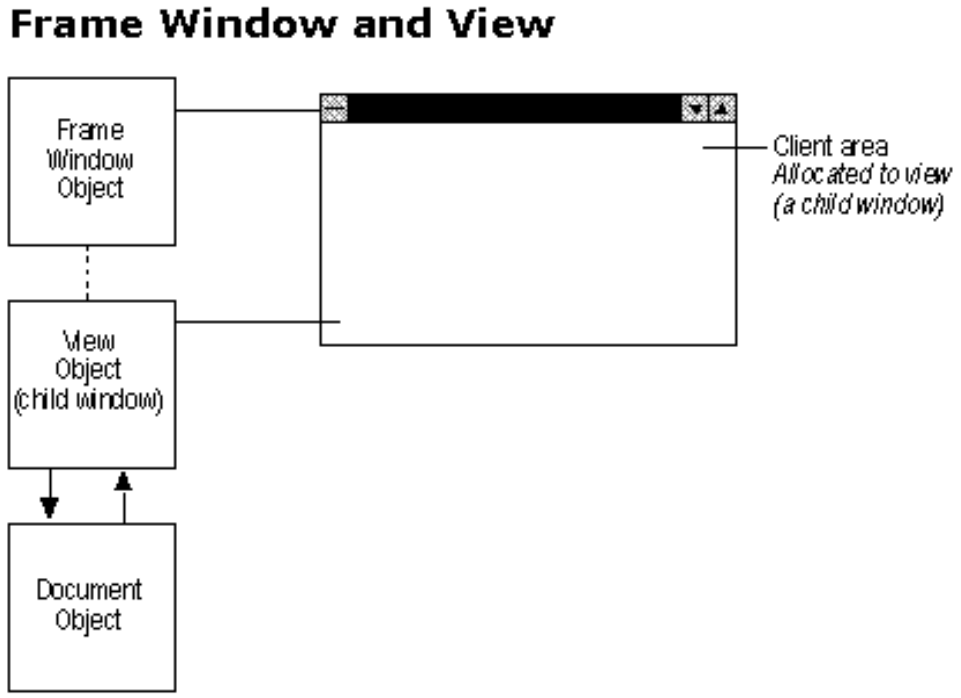
The **DECLARE_MESSAGE_MAP()** macro in CEXApp class declaration declares an array of message map entries as part of the class.

The BEGIN_MESSAGE_MAP and END_MESSAGE_MAP macro encloses initializations for this array that represent individual messages that the class can respond to.

```
BEGIN_MESSAGE_MAP(CEXApp, CWinApp)
        //{{AFX_MSG_MAP(CEXApp)
        ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG_MAP
        // Standard file based document commands
        ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
        ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
        // Standard print setup command
        ON_COMMAND(ID_FILE_PRINT_SETUP,
                CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

The Frame, Document and View:

MFC applications typically have at least two windows - the frame window and the view window. The also usually have a document object that is not a visual object but contains the data in the view window.



Frame Windows:

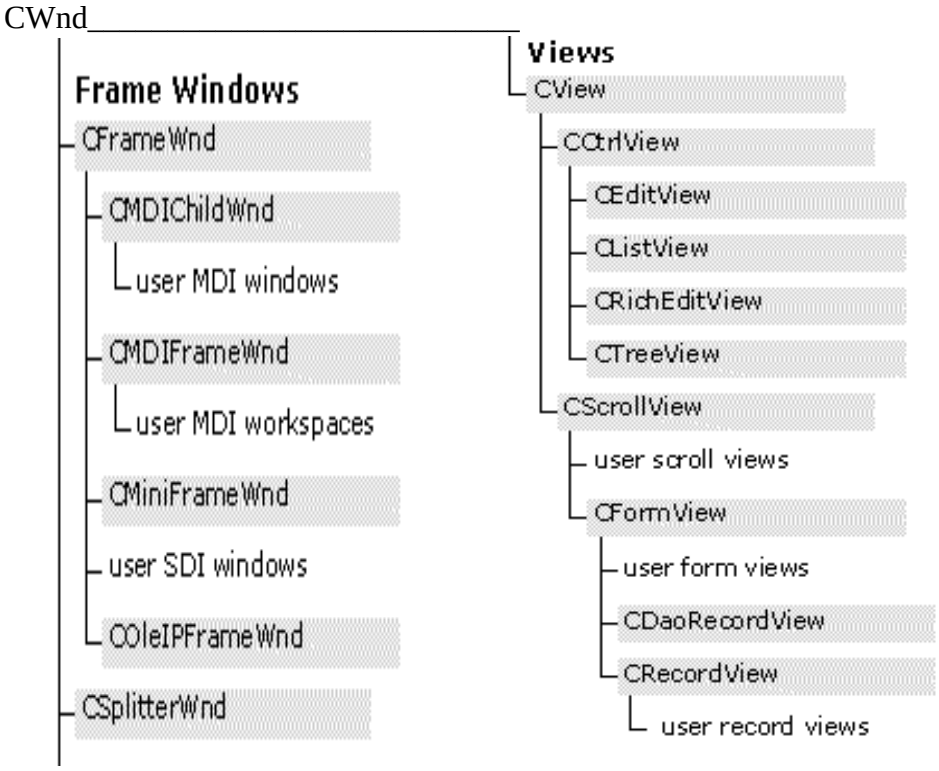
When an application runs under Microsoft Windows, the user interacts with documents displayed in frame windows. A document frame window has two major components: the frame and the contents that it frames. A document frame window can be a single document interface (SDI) frame window or a multiple document interface (MDI) child window. Windows manages most of the user's interaction with the frame window: moving and resizing the window, closing it, and minimizing and maximizing it. You manage the contents inside the frame.

Frame windows encapsulate the functionality of the application's main window and manage the application's **menu bar, toolbar buttons and status bar**.

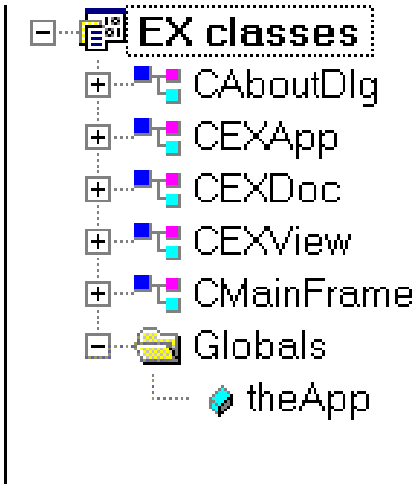
Another common arrangement is for the frame window to frame multiple views, usually using a **splitter** window. In a splitter window, the frame window's client area is occupied by a splitter window, which in turn has multiple child windows, called panes, which are views.

View Windows:

View windows present the contents of its documents to the user for interaction. The MFC framework uses frame windows to contain views. The two components—frame and contents—are represented and managed by two different classes in MFC. A frame-window class manages the frame, and a view class manages the contents. The view window is a child of the frame window. Drawing and other user interaction with the document take place in the view's client area, not the frame window's client area. The frame window provides a visible frame around a view, complete with a caption bar and standard window controls such as a control menu, buttons to minimize and maximize the window, and controls for resizing the window. The "contents" consist of the window's client area, which is fully occupied by a child window—the view. The following figure shows the relationship between a frame window and a view. View windows classes exist that support scrolling, text editing, list and tree controls, and dialog-like forms.



Example of Main Frame (MainFrm.h and MainFrm.cpp)



Here's the class declaration:

```
class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
        CMainFrame();
        DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMainFrame)
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CMainFrame();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif
```

Note these member variables. You can add additional control bars.

```
        protected:  // control bar embedded members
```

```
        CStatusBar  m_wndStatusBar;
        CToolBar    m_wndToolBar;

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

Here's the implementation.

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "EX.h"

#include "MainFrm.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        //{{AFX_MSG_MAP(CMainFrame)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code !
        ON_WM_CREATE()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
```

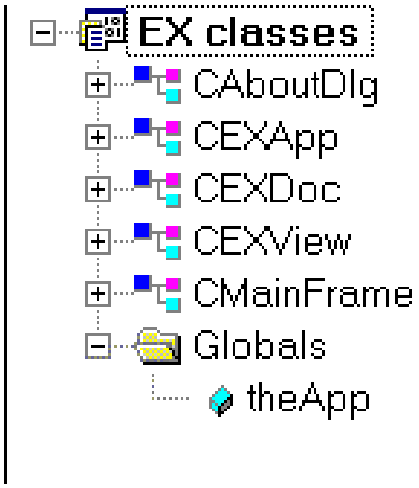
```
{
        ID_SEPARATOR,           // status line indicator
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
};

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
        // TODO: add member initialization code here

}

CMainFrame::~CMainFrame()
{
}
```

Here is where the toolbar and status bars are initialized.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;

        if (!m_wndToolBar.Create(this) ||
                !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
        {
                TRACE0("Failed to create toolbar\n");
                return -1;      // fail to create
        }

        if (!m_wndStatusBar.Create(this) ||
                !m_wndStatusBar.SetIndicators(indicators,
                  sizeof(indicators)/sizeof(UINT)))
        {
                TRACE0("Failed to create status bar\n");
                return -1;      // fail to create
        }

        // TODO: Remove this if you don't want tool tips or a resizeable toolbar
        m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
                CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

        // TODO: Delete these three lines if you don't want the toolbar to
```

```
	//  be dockable
```
The Microsoft Foundation Class Library supports dockable toolbars. A dockable toolbar can be attached, or docked, to any side of its parent window, or it can be detached, or floated, in its own mini-frame window.
```
	m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
	EnableDocking(CBRS_ALIGN_ANY);
	DockControlBar(&m_wndToolBar);

	return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs

	return CFrameWnd::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
	CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
	CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
```

Example of the Document Class (EXDoc.h and EXDoc.cpp):




```
class CEXDoc : public CDocument
{
protected: // create from serialization only
        CEXDoc();
```

DECLARE_DYNCREATE used instead of DECLARE_SERIAL because we will never need >> to retrieve a document. CDocument::OnOpenDocument explicitly.

```
        DECLARE_DYNCREATE(CEXDoc)

// Attributes
public:

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CEXDoc)
        public:
```

OnNewDocument called with File:New. For SDI this is called to re-initialize the only document. Things that might go in a constructor for MDI might go here for SDI.

```
        virtual BOOL OnNewDocument();
```

Serializer here. You must override this to load and save your document.

```
        virtual void Serialize(CArchive& ar);
        //}}AFX_VIRTUAL

// Implementation
```

```
public:
	virtual ~CEXDoc();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CEXDoc)
		// NOTE - the ClassWizard will add and remove member functions
here.
		//    DO NOT EDIT what you see in these blocks of generated code
!
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Here's the implementation.

```
#include "EX.h"

#include "EXDoc.h"

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

/////////////////////////////////////////////////////////////////////////////
// CEXDoc
```

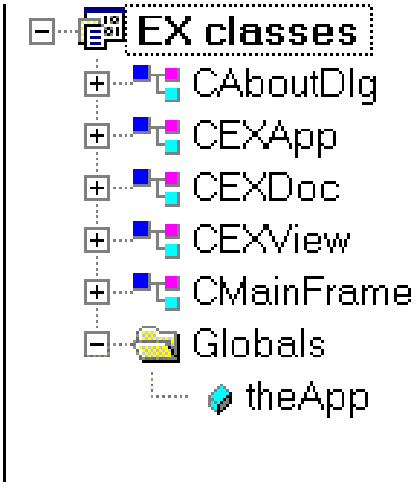
Note.

```
IMPLEMENT_DYNCREATE(CEXDoc, CDocument)

BEGIN_MESSAGE_MAP(CEXDoc, CDocument)
	//{{AFX_MSG_MAP(CEXDoc)
		// NOTE - the ClassWizard will add and remove mapping macros
here.
		//    DO NOT EDIT what you see in these blocks of generated
code!
	//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CEXDoc construction/destruction

CEXDoc::CEXDoc()
{
	// TODO: add one-time construction code here

}

CEXDoc::~CEXDoc()
{
}

BOOL CEXDoc::OnNewDocument()
{
	if (!CDocument::OnNewDocument())
		return FALSE;

	// TODO: add reinitialization code here
	// (SDI documents will reuse this document)

	return TRUE;
}



/////////////////////////////////////////////////////////////////////////////
// CEXDoc serialization

void CEXDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
}

/////////////////////////////////////////////////////////////////////////////
// CEXDoc diagnostics
```

```
#ifdef _DEBUG
void CEXDoc::AssertValid() const
{
	CDocument::AssertValid();
}

void CEXDoc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CEXDoc commands
```

The View Class (EXView.h and EXView.cpp):




```
class CEXView : public CView
{
protected: // create from serialization only
        CEXView();
        DECLARE_DYNCREATE(CEXView)

// Attributes
public:
        CEXDoc* GetDocument();

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CEXView)
        public:
```

OnDraw will almost always be overridden.. This baby actually draws your document detail in the view.

```
        virtual void OnDraw(CDC* pDC);  // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
```

Printing is enabled with these.

```
        virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
        virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
        //}}AFX_VIRTUAL
```

```
// Implementation
public:
	virtual ~CEXView();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CEXView)
	// NOTE - the ClassWizard will add and remove member functions here.
	//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Here's the implementation.

```
// EXView.cpp : implementation of the CEXView class
//

#include "stdafx.h"
#include "EX.h"

#include "EXDoc.h"
#include "EXView.h"

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

/////////////////////////////////////////////////////////////////////////////
// CEXView

IMPLEMENT_DYNCREATE(CEXView, CView)

BEGIN_MESSAGE_MAP(CEXView, CView)
	//{{AFX_MSG_MAP(CEXView)
		// NOTE - the ClassWizard will add and remove mapping macros
here.
```

```
                //    DO NOT EDIT what you see in these blocks of generated
code!
        //}}AFX_MSG_MAP
        // Standard printing commands
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW,
CView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CEXView construction/destruction

CEXView::CEXView()
{
        // TODO: add construction code here

}

CEXView::~CEXView()
{
}

BOOL CEXView::PreCreateWindow(CREATESTRUCT& cs)
{
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs

        return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CEXView drawing
```

OnDraw will need to do something.

```
void CEXView::OnDraw(CDC* pDC)
{
```

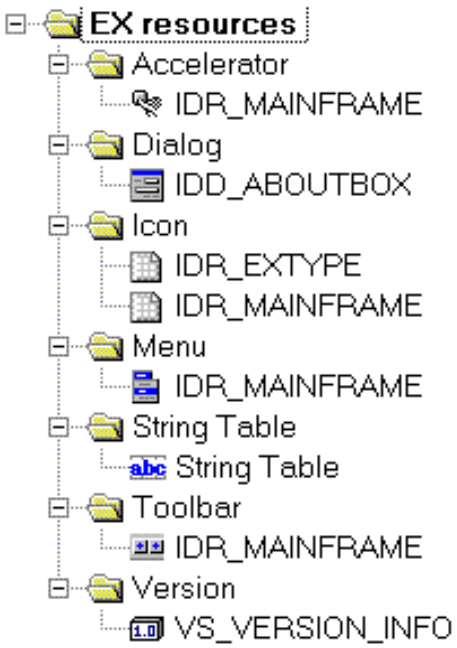
Here's the document.

```
        CEXDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        // TODO: add draw code for native data here
}

/////////////////////////////////////////////////////////////////////////////
// CEXView printing
```

```
BOOL CEXView::OnPreparePrinting(CPrintInfo* pInfo)
{
	// default preparation
	return DoPreparePrinting(pInfo);
}

void CEXView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
	// TODO: add extra initialization before printing
}

void CEXView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
	// TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////////
// CEXView diagnostics

#ifdef _DEBUG
void CEXView::AssertValid() const
{
	CView::AssertValid();
}

void CEXView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CEXDoc* CEXView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CEXDoc)));
	return (CEXDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CEXView message handlers
```

Example Resources ():

Open the resource view and, after opening each type, you will see:



Open each to see what is contained in each.

The Accelerator contains the keyboard shortcuts

The Dialog contains the About information.

There are two icons - IDR_MAINFRAME is the application icon. IDR_EXTYPE is the icon representing the application's document type.

The menu is the menu bar.

The string table contains all the strings used by the application. One of particular interest is IDR_MAINFRAME. This is also called the document template string. This string has up to 9 substrings representing:

<windowTitle>\n	Title on Main Frame
<docName>\n	Root document name (+ number)
<fileNewName>\n	Document type (when multiple)
<filterName>\n	Filter for files
<filterExt>\n	Extension used with file dialogs
<regFileTypeID>\n	File type registered in registry.
<regFileTypeName>\n	Name registered

In our application we have

Hello Fols\n
\n
EX\n
EX Files (*.gjk)\n
gjk\n
EX Document\n
EX Document\n

Toolbar has the application toolbar

Version number has the version information.

Running the Application:

As is: Just compile and run.

It does a lot - but nothing in the view window. Note the resources.

Let's do the following:

Document class

add a string member.
inititialize it from the string resource

View class

add code to display the string

1. Adding a string resource.

   Open the string resource and add the string (go to the bottom - there is an empty block. Start typing the value. Change the string name)

   IDS_DISPLAYVALUE This is a Test

2. Add a member variable to the Document class (in EXDoc.h).

```
class CEXDoc : public CDocument
{
protected: // create from serialization only
        CEXDoc();
        DECLARE_DYNCREATE(CEXDoc)

// Attributes
public:
        CString      m_sData;
```

3. Initialize this member variable (in EXDoc.cpp, OnNewDocument())

```
BOOL CEXDoc::OnNewDocument()
{
        if (!CDocument::OnNewDocument())
                return FALSE;
```

```
        // TODO: add reinitialization code here
        // (SDI documents will reuse this document)
        m_sData.LoadString( IDS_DISPLAYVALUE );
```

Note: LoadString Reads a Windows string resource, identified by nID, into an existing CString object.

```
        return TRUE;
}
```

4. Fill in the serialize operations (in EXDoc.cpp, Serialize()).

```
/////////////////////////////////////////////////////////////////////////////
// CEXDoc serialization

void CEXDoc::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {
                // TODO: add storing code here
                ar << m_sData;
        }
        else
        {
                // TODO: add loading code here
                ar >> m_sData;
        }
}
```

5. Modify the View (in EXView.cpp, OnDraw())

```
/////////////////////////////////////////////////////////////////////////////
// CEXView drawing

void CEXView::OnDraw(CDC* pDC)
{
        CEXDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        // TODO: add draw code for native data here
        CRect rect;
        GetClientRect( &rect );
        pDC->DPtoLP(&rect);
        pDC->DrawText( pDoc->m_sData, &rect,
```

```
                    DT_CENTER | DT_VCENTER | DT_SINGLELINE );
}
```

Run this. Save and open a file.

Some side notes:

CDC Class:

The CDC class defines a class of device-context objects. The CDC object provides member functions for working with a device context, such as a display or printer, as well as members for working with a display context associated with the client area of a window.

Do all drawing through the member functions of a CDC object. The class provides member functions for device-context operations, working with drawing tools, type-safe graphics device interface (GDI) object selection, and working with colors and palettes. It also provides member functions for getting and setting drawing attributes, mapping, working with the viewport, working with the window extent, converting coordinates, working with regions, clipping, drawing lines, and drawing simple shapes, ellipses, and polygons. Member functions are also provided for drawing text, working with fonts, using printer escapes, scrolling, and playing metafiles.

To use a CDC object, construct it, and then call its member functions that parallel Windows functions that use device contexts.

Note Under Windows 95, all screen coordinates are limited to 16 bits. Therefore, an int passed to a CDC member function must lie in the range –32768 to 32767.

GetClientRect(&rect);

CWnd::GetClientRect

void GetClientRect(LPRECT lpRect) const;

Parameters

lpRect Points to a RECT structure or a CRect object to receive the client coordinates. The left and top members will be 0. The right and bottom members will contain the width and height of the window.

Remarks

Copies the client coordinates of the CWnd client area into the structure pointed to by lpRect. The client coordinates specify the upper-left and lower-right corners of the client area. Since client coordinates are relative to the upper-left corners of the CWnd client area, the coordinates of the upper-left corner are (0,0).

pDC->DPtoLP(&rect);

Converts device units into logical units. The function maps the coordinates of each point, or dimension of a size, from the device coordinate system into GDI's logical coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents for the device's window and viewport.