**Unit-2**

**Chapter-13:**

**Menus, Keyboard Accelerators, the Rich Edit Control, and Property Sheets**

**Q: Explain windows menu and keyboard accelerators?**

**Ans:**

**Windows Menus:**

1. A Microsoft Windows menu is a familiar application element that consists of a top-level horizontal list of items with associated pop-up menus that appear when the user selects a top-level item.
2. Most of the time, you define for a frame window a default menu resource that loads when the window is created. You can also define a menu resource independent of a frame window. In that case, your program must call the functions necessary to load and activate the menu.
3. A menu resource completely defines the initial appearance of a menu. Menu items can be grayed or have check marks, and bars can separate groups of menu items.
4. Multiple levels of pop-up menus are possible. If a first-level menu item is associated with a subsidiary pop-up menu, the menu item carries a right-pointing arrow symbol.
5. Visual C++ includes an easy-to-use menu-resource editing tool. This tool lets you edit menus in a wysiwyg

environment.

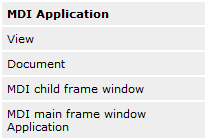
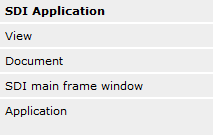
1. Each menu item has a properties dialog that defines all the characteristics of that item. The resulting resource definition is stored in the application's resource script (RC) file. Each menu item is associated with an ID, such as *ID\_FILE\_OPEN*, that is defined in the resource.h file.
2. The MFC library extends the functionality of the standard menus for Windows. Each menu item can have a prompt string that appears in the frame's status bar when the item is highlighted.

**Keyboard Accelerators:**

1. You've probably noticed that most menu items contain an underlined letter. In Visual C++ (and most other applications), pressing Alt-F followed by S activates the File Save menu item.
2. This shortcut system is the standard Windows method of using the keyboard to choose commands from menus. If you look at an application's menu resource script (or the menu editor's properties dialog), you will see an ampersand (&) preceding the character that is underlined in each of the application's menu items.
3. Windows offers an alternative way of linking keystrokes to menu items. The keyboard accelerator resource consists of a table of key combinations with associated command IDs.
4. The Edit Copy menu item (with command ID *ID\_EDIT\_COPY*), for example, might be linked to the Ctrl-C key combination through a keyboard accelerator entry.
5. A keyboard accelerator entry does not have to be associated with a menu item. If no Edit Copy menu item were present, the Ctrl-C key combination would nevertheless activate the *ID\_EDIT\_COPY* command.

# Command Processing:

Most command messages originate in the application's frame window, and without the application framework in the picture, that's where you would put the message handlers. With command routing, however, you can handle a message almost anywhere. When the application framework sees a frame window command message, it starts looking for message handlers in one of the sequences listed here.



Most applications have a particular command handler in only one class, but suppose your one-view application has an identical handler in both the view class and the document class. Because the view is higher in the command route, only the view's command handler function will be called.

## Update Command User Interface Handlers:

You often need to change the appearance of a menu item to match the internal state of your application. You've undoubtedly seen such grayed menu items in Windows-based applications, and you've probably also seen check marks next to menu items.

Update command UI handlers apply only to items on pop-up menus, not to top-level menu items that are permanently displayed.

You cannot use UI handler to disable the file menu item.

# MFC Text Editing Options:

Windows itself supplies two text editing tools: edit control and Windows rich edit common control. Both can be used as controls within dialogs, but both can also be made to look like view windows. The MFC library supports this versatility with the *CEditView* and *CRichEditView* classes.

## The *CEditView* Class

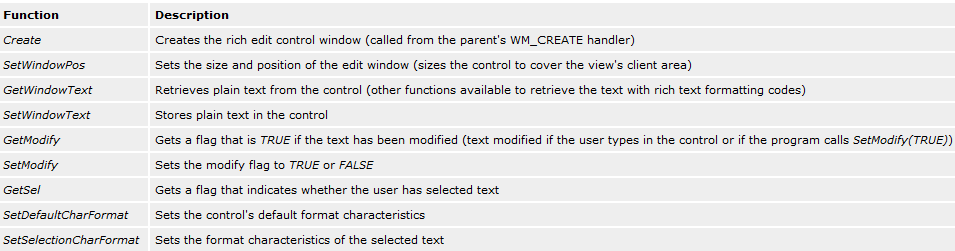
1. This class is based on the Windows edit control, so it inherits all the edit control's limitations.
2. Text size is limited to 64 KB, and you can't mix fonts.
3. AppWizard gives you the option of making *CEditView* the base class of your view class.
4. The *CEditView* class implements and maps the clipboard cut, copy, and paste functions, so they appear active on the Edit menu.

## The *CRichEditView* Class

1. This class uses the rich edit control, so it supports mixed formats and large quantities of text.
2. The *CRichEditView* class is designed to be used with the *CRichEditDoc* and *CRichEditCntrItem* classes to implement a complete ActiveX container application.

## The *CRichEditCtrl* Class

1. This class wraps the rich edit control, and you can use it to make a fairly decent text editor.
2. We'll use an ordinary view class derived from *CView*, and we'll cover the view's client area with a big rich edit control that resizes itself when the view size changes.
3. The *CRichEditCtrl* class has dozens of useful member functions, and it picks up other functions from its *CWnd* base class..



**Q: Write a short note on Property Sheet?**

**Ans:**

**Property Sheet:**

1. A property sheet is a nice UI element that allows you to cram lots of categorized information into a small dialog.
2. The user selects pages by clicking on their tabs. Windows offers a tab control that you can insert in a dialog, but it's more likely that you'll want to put dialogs inside the tab control.
3. The MFC library supports this, and the result is called a property sheet. The individual dialogs are called property pages.

## Building a Property Sheet

Follow these general steps to build a property sheet using the Visual C++ tools:

1. Use the resource editor to create a series of dialog templates that are all approximately the same size. The captions are the strings that you want to display on the tabs.
2. Use ClassWizard to generate a class for each template. Select *CPropertyPage* as the base class. Add data members for the controls.
3. Use ClassWizard to generate a single class derived from *CPropertySheet*.
4. To the sheet class, add one data member for each page class.
5. In the sheet class constructor, call the *AddPage* member function for each page, specifying the address of the embedded page object.
6. In your application, construct an object of the derived *CPropertySheet* class, and then call *DoModal*. You must specify a caption in the constructor call, but you can change the caption later by calling *CPropertySheet::SetTitle*.
7. Take care of programming for the Apply button.

**Chapter-14:**

**Toolbars and Status Bars**

**Q: Write a short note on Toolbar and Status bar?**

**Ans:**

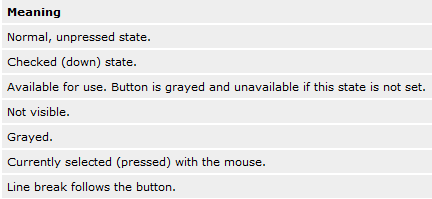
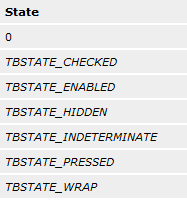
# The Toolbar:

1. Tool bar is an object of class CToolBar and this class is derived from class CControlBar which is itself derived from CWnd.
2. CControlBar supports control bar window that are process inside framw window. This control bar window resize and reposition themselves. As the parent frame move can change size.
3. Application frame work take care of construction, window creation and destruction of control bar object.
4. A toolbar consists of a number of horizontally (or vertically) arranged graphical buttons that might be clustered in groups. The programming interface determines the grouping.
5. The graphical images for the buttons are stored in a single bitmap that is attached to the application's resource file. When a button is clicked, it sends a command message, as do menus and keyboard accelerators.
6. An update command UI message handler is used to update the button's state, which in turn is used by the application framework to modify the button's graphical image.



## Button States:

Each button can assume the following states.



A button can behave in either of two ways: it can be a pushbutton, which is down only when currently selected by the mouse, or it can be a check box button, which can be toggled up and down with mouse clicks. All buttons in the standard application framework toolbar are pushbuttons.

# ToolTips:

When the user positions the mouse on a toolbar button for a certain interval, text is displayed in a little ToolTip box next to the button. To create a ToolTip, you simply add the tip text to the end of the menu prompt, preceded by a newline (\n) character. The resource editor lets you edit the prompt string while you are editing the toolbar images. Just double-click in the left panel.

# The Status Bar:

# Status bar is an object of class CStatusBar and this class is derived from class CControlBar which is itself derived from CWnd.

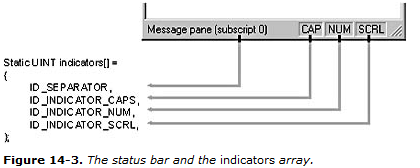
# The status bar window neither accepts user input nor generates command messages. Its job is simply to display text in panes under program control.

# The status bar supports two types of text panes—message line panes and status indicator panes.

# To use the status bar for application-specific data, you must first disable the standard status bar that displays the menu prompt and key-board status.

## The Status Bar Definition:

The static *indicators* array that AppWizard generates in the MainFrm.cpp file defines the panes for the application's status bar. The constant *ID\_SEPARATOR* identifies a message line pane; the other constants are string resource IDs that identify indicator panes. Figure 14-3 shows the *indicators* array and its relationship to the standard framework status bar.



The *CStatusBar::SetIndicators* member function, called in the application's derived frame class, configures the status bar according to the contents of the *indicators* array.

## The Message Line:

A message line pane displays a string that the program supplies dynamically. To set the value of the message line, you must first get access to the status bar object and then you must call the *CStatusBar::SetPaneText* member function with a zero-based index parameter. Pane 0 is the leftmost pane, 1 is the next pane to the right, and so forth.

The following code fragment is part of a view class member function. Note that you must navigate up to the application object and then back down to the main frame window.

CMainFrame\* pFrame = (CMainFrame\*) AfxGetApp()->m\_pMainWnd;

CStatusBar\* pStatus = &pFrame->m\_wndStatusBar;

pStatus->SetPaneText(0, "message line for first pane");

Normally, the length of a message line pane is exactly one-fourth the width of the display. If, however, the message line is the first (index 0) pane, it is a stretchy pane without a beveled border. Its minimum length is one-fourth the display width, and it expands if room is available in the status bar.

## The Status Indicator:

A status indicator pane is linked to a single resource-supplied string that is displayed or hidden by logic in an associated update command UI message handler function. An indicator is identified by a string resource ID, and that same ID is used to route update command UI messages. The Caps Lock indicator is handled in the frame class by a message map entry and a handler function equivalent to those shown below. The *Enable* function turns on the indicator if the Caps Lock mode is set.

ON\_UPDATE\_COMMAND\_UI(ID\_INDICATOR\_CAPS, OnUpdateKeyCapsLock)

void CMainFrame::OnUpdateKeyCapsLock(CCmdUI\* pCmdUI)

{

pCmdUI->Enable(::GetKeyState(VK\_CAPITAL) & 1);

}

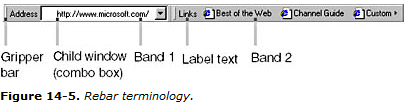
The status bar update command UI functions are called during idle processing so that the status bar is updated whenever your application receives messages. The length of a status indicator pane is the exact length of the corresponding resource string.

# The Internet Explorer Rebar Toolbar:

The rebar differs from the default MFC toolbar in that it provides grippers and allows the user to "slide" its horizontal and vertical positions, whereas the MFC toolbar's position is changed via drag-and-drop docking.

Rebars also allow the developer to provide many more internal control types—such as drop-down menus—than are available in *CToolBar*.

Figure 14-5 shows the various terminology used on a rebar. Each internal toolbar in a rebar is called a band. The raised edge where the user slides the band is called a gripper. Each band can also have a label.



MFC provides two classes that facilitate working with rebars:

* ***CReBar***—A high-level abstraction class that provides members for adding *CToolBar* and *CDialogBar* classes to rebars as bands. *CReBar* also handles communication (such as message notifications) between the underlying control and the MFC framework.
* ***CReBarCtrl***—A low-level wrapper class that wraps the IE ReBar control. This class provides numerous members for creating and manipulating rebars but does not provide the niceties that are found in *CReBar*.

Most MFC applications use *CReBar* and call the member function *GetReBarCtrl*, which returns a *CReBarCtrl* pointer to gain access to the lower-level control if needed.

**Chapter-15:**

**A Reusable Frame Window Base Class**

**Q: Why reusable base class are difficult to write?**

**Ans:**

1. In a normal application, you write code for software components that solve particular problems. With reusable base classes, you must anticipate future programming needs, both your own and those of others.
2. You have to write a class that is general and complete yet efficient and easy to use.
3. Window base program remember whether they had minimize to task bar or whether they have been maximize to full screen.

**Q: Write a short note on CPersistentFrame class?**

**Ans:**

**The *CPersistentFrame* Class**

*CPersistentFrame* that is derived from the *CFrameWnd* class. This *CPersistentFrame* class supports a persistent SDI (Single Document Interface) frame window that remembers the following characteristics.

* Window size
* Window position
* Maximized status
* Minimized status
* Toolbar and status bar enablement and position

When you terminate an application that's built with the *CPersistentFrame* class, the above information is saved on disk in the Windows Registry. When the application starts again, it reads the Registry and restores the frame to its state at the previous exit.

# The *CFrameWnd* Class and the *ActivateFrame* Member Function:

1. In an MFC SDI application, the main frame window is always the parent of the view window. This frame window is created first, and then the control bars and the view are created as child windows. The application framework ensures that the child windows shrink and expand appropriately as the user changes the size of the frame window. It wouldn't make sense to change the view size after the frame was created.
2. The key to controlling the frame's size is the *CFrameWnd::ActivateFrame* member function.
3. The application framework calls this virtual function (declared in *CFrameWnd*) during the SDI main frame window creation process (and in response to the File New and File Open commands).
4. The framework's job is to call the *CWnd::ShowWindow* function with the parameter *nCmdShow*. *ShowWindow* makes the frame window visible along with its menu, view window, and control bars. The *nCmdShow* parameter determines whether the window is maximized or minimized or both.
5. If you override *ActivateFrame* in your derived frame class, you can change the value of *nCmdShow* before passing it to the *CFrameWnd::ActivateFrame* function.
6. You can also call the *CWnd::SetWindowPlacement* function, which sets the size and position of the frame window, and you can set the visible status of the control bars. Because all changes are made before the frame window becomes visible, no annoying flash occurs on the screen.
7. You must be careful not to reset the frame window's position and size after every File New or File Open command. A first-time flag data member ensures that your *CPersistentFrame::ActivateFrame* function operates only when the application starts.

# The *PreCreateWindow* Member Function:

1. *PreCreateWindow*, declared at the *CWnd* level, is another virtual function that you can override to change the characteristics of your window before it is displayed.
2. The framework calls this function before it calls *ActivateFrame*. AppWizard always generates an overridden *PreCreateWindow* function in your project's view and frame window classes.
3. This function has a *CREATESTRUCT* structure as a parameter, and two of the data members in this structure are *style* and *dwExStyle*.
4. You can change these data members before passing the structure on to the base class *PreCreateWindow* function. The *style* flag determines whether the window has a border, scroll bars, a minimize box, and so on.
5. The *dwExStyle* flag controls other characteristics, such as always-on-top status. See the online documentation for Window Styles and Extended Window Styles for details.
6. The *CREATESTRUCT* member *lpszClass* is also useful to change the window's background brush, cursor, or icon. It makes no sense to change the brush or cursor in a frame window because the view window covers the client area.

# Using the *CString* Class:

The MFC *CString* class is a significant de facto extension to the C++ language. As the *Microsoft Foundation* *Classes and Templates* section of the online help points out, the *CString* class has many useful operators and member functions, but perhaps its most important feature is its dynamic memory allocation. You never have to worry about the size of a *CString* object. The statements here represent typical uses of *CString* objects.

CString strFirstName("Elvis");

CString strLastName("Presley");

CString strTruth = strFirstName + " " + strLastName; // concatenation

strTruth += " is alive";

ASSERT(strTruth == "Elvis Presley is alive");

ASSERT(strTruth.Left(5) == strFirstName);

ASSERT(strTruth[2] == `v'); // subscript operator

**Chapter-16:**

# Separating the Document from Its View

**Q: What are the functions that interact with view and document?**

**Ans:**

* 1. The document object holds the data and that the view object displays the data and allows editing.
  2. A complex handshaking process takes place among the document, the view, and the rest of the application framework.
  3. Two are nonvirtual base class functions that you call in your derived classes; three are virtual functions that you often override in your derived classes.

## The *CView::GetDocument* Function

A view object has one and only one associated document object. The *GetDocument* function allows an application to navigate from a view to its document. Suppose a view object gets a message that the user has entered new data into an edit control. The view must tell the document object to update its internal data accordingly.

This function is an inline function, and it looks something like this:

CMyDoc\* GetDocument()

{

return (CMyDoc\*)

m\_pDocument;

}

When the compiler sees a call to *GetDocument* in your view class code, it uses the derived class version instead of the *CDocument* version, so you do not have to cast the returned pointer to your derived document class. Because the *CView::GetDocument* function is not a virtual function, a statement such as

pView->GetDocument(); // pView is declared CView\*

calls the base class *GetDocument* function and thus returns a pointer to a *CDocument* object.

## The *CDocument::UpdateAllViews* Function

If the document data changes for any reason, all views must be notified so that they can update their representations of that data. If *UpdateAllViews* is called from a member function of a derived document class, its first parameter, *pSender*, is *NULL*. If *UpdateAllViews* is called from a member function of a derived view class, set the *pSender* parameter to the current view, like this:

GetDocument()->UpdateAllViews(this);

## The *CView::OnUpdate* Function

This virtual function is called by the application framework in response to your application's call to the *CDocument::UpdateAllViews* function. *OnUpdate* function accesses the document, gets the document's data, and then updates the view's data members or controls to reflect the changes. The *OnUpdate* function might look something like this:

void CMyView::OnUpdate(CView\* pSender, LPARAM lHint, CObject\* pHint)

{

CMyDocument\* pMyDoc = GetDocument();

CString lastName = pMyDoc->GetLastName();

m\_pNameStatic->SetWindowText(lastName); // m\_pNameStatic is

// a CMyView data member

}

## The *CView::OnInitialUpdate* Function

This virtual *CView* function is called when the application starts, when the user chooses New from the File menu, and when the user chooses Open from the File menu. The *CView* base class version of *OnInitialUpdate* does nothing but call *OnUpdate*. If you override *OnInitialUpdate* in your derived view class, be sure that the view class calls the base class's *OnInitialUpdate* function or the derived class's *OnUpdate* function.

You can use your derived class's *OnInitialUpdate* function to initialize your view object. When the application starts, the application framework calls *OnInitialUpdate* immediately after *OnCreate* (if you've mapped *OnCreate* in your view class). *OnCreate* is called once, but *OnInitialUpdate* can be called many times.

## The *CDocument::OnNewDocument* Function

The framework calls this virtual function after a document object is first constructed and when the user chooses New from the File menu in an SDI application. This is a good place to set the initial values of your document's data members. AppWizard generates an overridden *OnNewDocument* function in your derived document class. Be sure to retain the call to the base class function.

# The *CDocument::DeleteContents* Function

This function is used to delete the contents of your document. The application framework declares a virtual *DeleteContents* function for the *CDocument* class. The application framework calls your overridden *DeleteContents* function when the document is closed.

**Chapter-17:**

# Reading and Writing Documents—SDI Applications

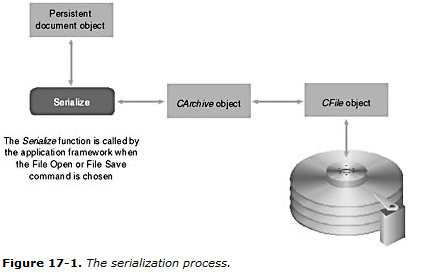
**Q: Write a short note on Serialization?**

**Ans:**

1. The objects can be persistent, which means they can be saved on disk when a program exits and then can be restored when the program is restarted. This process of saving and restoring objects is called serialization.
2. In the Microsoft Foundation Class (MFC) library, designated classes have a member function named *Serialize*.

## Disk Files and Archives:

1. A *CFile* object encapsulates the binary file handle that you get through the Win32 function *CreateFile*.
2. The application framework uses this file handle for Win32 *ReadFile*, *WriteFile*, and *SetFilePointer* calls.
3. If your application does no direct disk I/O but instead relies on the serialization process, you can avoid direct use of *CFile* objects. Between the *Serialize* function and the *CFile* object is an archive object (of class *CArchive*), as shown in Figure 17-1.
4. The *CArchive* object buffers data for the *CFile* object, and it maintains an internal flag that indicates whether the archive is storing (writing to disk) or loading (reading from disk). Only one active archive is associated with a file at any one time.
5. The application framework takes care of constructing the *CFile* and *CArchive* objects, opening the disk file for the *CFile* object and associating the archive object with the file.
6. The application framework calls the document's *Serialize* function during the File Open and File Save processes.



## Making a Class Serializable:

A serializable class must be derived directly or indirectly from *CObject*. In addition (with some exceptions), the class declaration must contain the *DECLARE\_SERIAL* macro call, and the class implementation file must con- tain the *IMPLEMENT\_SERIAL* macro call.

## Writing a *Serialize* Function:

*Serialize* is a virtual member function of class *CObject*, you must be sure that the return value and parameter types match the *CObject* declaration. The *Serialize* function for the *CStudent* class is below.

void CStudent::Serialize(CArchive& ar)

{

TRACE("Entering CStudent::Serialize\n");

if (ar.IsStoring()) {

ar << m\_strName << m\_nGrade;

}

else {

ar >> m\_strName >> m\_nGrade;

}

}

Most serialization functions call the *Serialize* functions of their base classes. If *CStudent* were derived from *CPerson*, for example, the first line of the *Serialize* function would be

CPerson::Serialize(ar);

The *Serialize* function for *CObject* (and for *CDocument*, which doesn't override it) doesn't do anything useful, so there's no need to call it.

**Q: Write a short note on Document template class?**

**Ans:**

## The Document Template Class:

The *InitInstance* function that AppWizard generates for your derived application class, you'll see that the following statements are featured:

CSingleDocTemplate\* pDocTemplate;

pDocTemplate = new CSingleDocTemplate(

IDR\_MAINFRAME,

RUNTIME\_CLASS(CStudentDoc),

RUNTIME\_CLASS(CMainFrame), // main SDI frame window

RUNTIME\_CLASS(CStudentView));

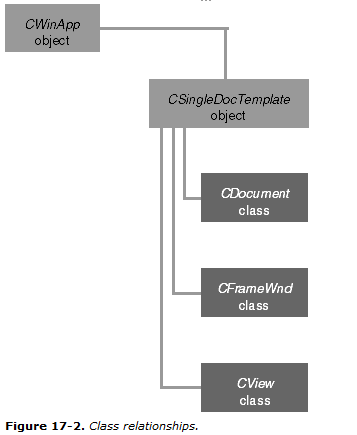
AddDocTemplate(pDocTemplate);

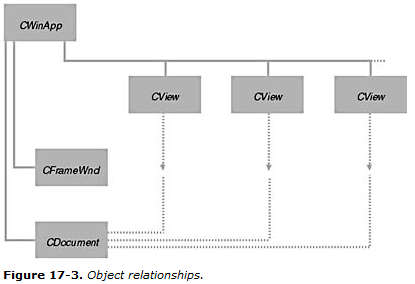
The *CSingleDocTemplate* class applies only to SDI applications because SDI applications are limited to one document object. *AddDocTemplate* is a member function of class *CWinApp*. The *AddDocTemplate* call, together with the document template constructor call, establishes the relationships among classes—the application class, the document class, the view window class, and the main frame window class.

The *DECLARE\_DYNCREATE* and *IMPLEMENT\_DYNCREATE* macros in a class declaration and implementation enable the MFC library to construct objects of the specified class dynamically.

With the template system, all that's required in your application class is use of the *RUNTIME\_CLASS* macro. Notice that the target class's declaration must be included for this macro to work. Figure 17-2 illustrates the relationships among the various classes, and Figure 17-3 illustrates the object relationships.

An SDI application can have only one template (and associated class groups), and when the SDI program is running, there can be only one document object and only one main frame window object.





**Chapter-18:**

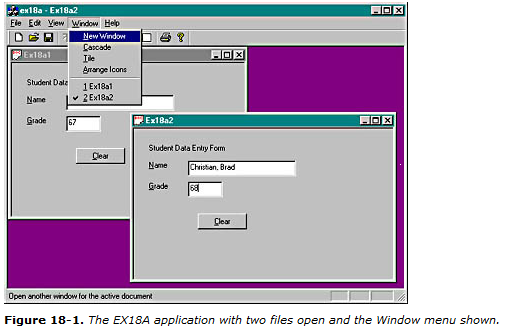
# Reading and Writing Documents—MDI Applications

# The MDI Application

Before you look at the MFC library code for MDI applications, you should be familiar with the operation of Microsoft Windows MDI programs. Take a close look at Visual C++ now. It's an MDI application whose "multiple documents" are program source code files. Visual C++ is not the most typical MDI application, however, because it collects its documents into projects. It's better to examine Microsoft Word or, better yet, a real MFC library MDI application—the kind that AppWizard generates.

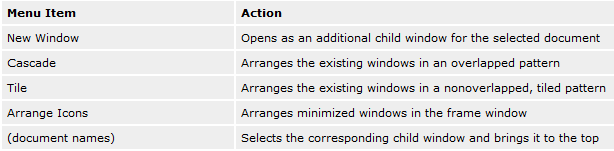
## A Typical MDI Application, MFC Style

This chapter's example, EX18A, is an MDI version of EX17A. Run the EX17A example to see an illustration of the SDI version after the user has selected a file. Now look at the MDI equivalent, as shown in Figure 18-1.



The user has two separate document files open, each in a separate MDI child window, but only one child window is active—the lower window, which lies on top of the other child window. The application has only one menu and one toolbar, and all commands are routed to the active child window. The main window's title bar reflects the name of the active child window's document file.

The child window's minimize box allows the user to reduce the child window to an icon in the main window. The application's Window menu (shown in Figure 18-1) lets the user control the presentation through the following items.



**Q: Write a short note on MDI Document Template Class?**

**Ans:**

## The MDI Document Template Class:

The MDI template construction call in *InitInstance* looks like this:

CMultiDocTemplate\* pDocTemplate;

pDocTemplate = new CMultiDocTemplate(

IDR\_EX18ATYPE,

RUNTIME\_CLASS(CStudentDoc),

RUNTIME\_CLASS(CChildFrame), // custom MDI child frame

RUNTIME\_CLASS(CStudentView));

AddDocTemplate(pDocTemplate);

An MDI application can use multiple document types and allows the simultaneous existence of more than one document object. This is the essence of the MDI application.

The single *AddDocTemplate* call shown above permits the MDI application to support multiple child windows, each connected to a document object and a view object. It's also possible to have several child windows (and corresponding view objects) connected to the same document object.

**Chapter-19:**

# Printing and Print Preview

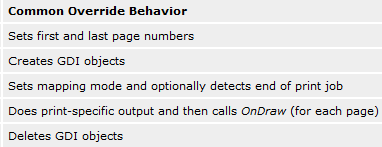
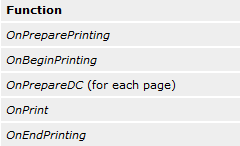
## The *CView::OnPrint* Function

The base class *OnPrint* function calls *OnDraw* and that *OnDraw* can use both a display device context and a printer device context. The mapping mode should be set before *OnPrint* is called. You can override *OnPrint* to print items that you don't need on the display, such as a title page, headers, and footers. The *OnPrint* parameters are as follows:

* A pointer to the device context
* A pointer to a print information object (*CPrintInfo*) that includes page dimensions, the current page number, and the maximum page number

The application framework calls the *OnPrint* function once for each page to be printed, with the current page number in the *CPrintInfo* structure.

The following table summarizes the important overridable *CView* print loop functions.



**Chapter-20:**

# Splitter Windows and Multiple Views

**Q: Write a short note on Splitter Window?**

**Ans:**

# The Splitter Window:

1. A splitter window appears as a special type of frame window that holds several views in panes.
2. The application can split the window on creation, or the user can split the window by choosing a menu command or by dragging a splitter box on the window's scroll bar.
3. After the window has been split, the user can move the splitter bars with the mouse to adjust the relative sizes of the panes.
4. Splitter windows can be used in both SDI and MDI applications.
5. An object of class *CSplitterWnd* represents the splitter window.
6. *CSplitterWnd* object is an actual window that fully occupies the frame window (*CFrameWnd* or *CMDIChildWnd*) client area.

There are two types of splitter windows:

1. Dynamic Splitter Window,
2. Static Splitter Window.

# Dynamic and Static Splitter Windows:

1. A dynamic splitter window allows the user to split the window at any time by choosing a menu item or by dragging a splitter box located on the scroll bar.
2. The panes in a dynamic splitter window generally use the same view class.
3. The top left pane is initialized to a particular view when the splitter window is created.
4. In a dynamic splitter window, scroll bars are shared among the views.
5. In a window with a single horizontal split, for example, the bottom scroll bar controls both views. A dynamic splitter application starts with a single view object.
6. When the user splits the frame, other view objects are constructed. When the user unsplits the frame, view objects are destroyed.
7. **Static Splitter Window:**
8. The panes of a static splitter window are defined when the window is first created and they cannot be changed.
9. The user can move the bars but cannot unsplit or resplit the window.
10. Static splitter windows can accommodate multiple view classes, with the configuration set at creation time.
11. In a static splitter window, each pane has separate scroll bars.
12. In a static splitter window application, all view objects are constructed when the frame is constructed and they are all destroyed when the frame is destroyed.

* **View Options:**

When you combine multiview presentation methods with application models, you get a number of permutations. Here are some of them:

* **SDI application with splitter window, single view class** This chapter's first example, EX20A, covers this case. Each splitter window pane can be scrolled to a different part of the document. The programmer determines the maximum number of horizontal and vertical panes; the user makes the split at runtime.
* **SDI application with splitter window, multiple view classes** The EX20B example illustrates this case. The programmer determines the number of panes and the sequence of views; the user can change the pane size at runtime.
* **SDI application with no splitter windows, multiple view classes** The EX20C example illustrates this case. The user switches view classes by making a selection from a menu.
* **MDI application with no splitter windows, single view class** This is the standard MDI application you've seen already in [Chapter 18](about:blank). The New Window menu item lets the user open a new child window for a document that's open already.
* **MDI application with no splitter windows, multiple view classes** A small change to the standard MDI application allows the use of multiple views. As example EX20D shows, all that's necessary is to add a menu item and a handler function for each additional view class available.
* **MDI application with splitter child windows** This case is covered thoroughly in the online documentation. The SCRIBBLE example illustrates the splitting of an MDI child window.

**Chapter-21:**

**Context-Sensitive Help**

**The Application Framework and WinHelp**

The application framework and WinHelp cooperate to give you context-sensitive help. Here are some of the main elements:

1. You select the Context-Sensitive Help option when you run AppWizard.
2. AppWizard generates a Help Topics item on your application's Help menu, and it creates one or more generic RTF files together with an HPJ file and a batch file that runs the Help Compiler.
3. AppWizard inserts a keyboard accelerator for the F1 key, and it maps the F1 key and the Help Topics menu item to member functions in the main frame window object.
4. When your program runs, it calls WinHelp when the user presses F1 or chooses the Help Topics menu item, passing a context ID that determines which help topic is displayed.

**Chapter-22:**

**Dynamic Link Libraries**

A DLL is a file an disk consists of global data, compile function and resources that become part of your process. It is compiled to load at a prepared base address. DLL has various exported functions and client program import those functions. Windows match the import and export wants its load the DLL.

## How Imports Are Matched to Exports

1. A DLL contains a table of exported functions. These functions are identified to the outside world by their symbolic names and (optionally) by integers called ordinal numbers.
2. The function table also contains the addresses of the functions within the DLL.
3. When the client program first loads the DLL, it doesn't know the addresses of the functions it needs to call, but it does know the symbols or ordinals.
4. The dynamic linking process then builds a table that connects the client's calls to the function addresses in the DLL.
5. In the DLL code you must explicitly declare your exported finction like this

\_ \_ declspace (dllexport) int MyFunction(int n);

1. On the client side you need to declare the corresponding import like this

\_ \_ declspace (dllimport) int MyFunction(int n);

* **Implicit Linkage vs. Explicit Linkage**

**Implicit Linkage:**

1. When you build a DLL, the linker produces a companion import LIB file, which contains every DLL's exported symbols and (optionally) ordinals, but no code.
2. The LIB file is a surrogate for the DLL that is added to the client program's project.
3. When you build (statically link) the client, the imported symbols are matched to the exported symbols in the LIB file, and those symbols (or ordinals) are bound into the EXE file.
4. The LIB file also contains the DLL filename (but not its full pathname), which gets stored inside the EXE file. When the client is loaded, Windows finds and loads the DLL and then dynamically links it by symbol or by ordinal.

**Explicit Linkage:**

1. Explicit linking is more appropriate for interpreted languages such as Microsoft Visual Basic.
2. With explicit linking, you don't use an import file; instead, you call the Win32 *LoadLibrary* function, specifying the DLL's pathname as a parameter.
3. With explicit linkage you can determine when DLL are loaded and unloaded.
4. It allows you to determine run time which DLL to load.

* **Symbolic Linkage vs. Ordinal Linkage**

1. The DLL version of the MFC library, however, uses ordinal linkage.
2. Ordinal linkage permits that program's EXE file to be smaller because it does not have to contain the long symbolic names of its imports.
3. If you build your own DLL with ordinal linkage, you must specify the ordinals in the project's DEF file, which doesn't have too many other uses in the Win32 environment.
4. If your exports are C++ functions, you must use decorated names in the DEF file (or declare your functions with *extern "C"*)*.* Here's a short extract from one of the MFC library DEF files:

?ReadList@CRecentFileList@@UAEXXZ @ 5458 NONAME

The numbers after the at (@) symbols are the ordinals.

# MFC DLLs—Extension vs. Regular

AppWizard lets you build two kinds of DLLs with MFC library support: extension DLLs and regular DLLs.

**Extension DLL:**

1. An extension DLL supports a C++ interface.
2. In other words, the DLL can export whole classes and the client can construct objects of those classes or derive classes from them.
3. An extension DLL dynamically links to the code in the DLL version of the MFC library.
4. An extension DLL requires that your client program be dynamically linked to the MFC library (the AppWizard default) and that both the client program and the extension DLL be synchronized to the same version of the MFC DLLs.
5. Extension DLLs are quite small; you can build a simple extension DLL with a size of 10 KB, which loads quickly.

**Regular DLL:**

1. The regular DLL can export only C-style functions.
2. It can't export C++ classes, member functions, or overloaded functions because every C++ compiler has its own method of decorating names.

# A Custom Control DLL

The original custom controls were written in pure C and configured as stand-alone DLLs. A regular DLL is the best choice for a custom control because the control doesn't need a C++ interface and because it can be used by any development system that accepts custom controls (such as the Borland C++ compiler). You'll probably want to use the MFC dynamic linking option because the resulting DLL will be small and quick to load.

## What Is a Custom Control?

The custom control acts like an ordinary control, such as the edit control, in that it sends WM\_COMMAND notification messages to its parent window and receives user-defined messages. The dialog editor lets you position custom controls in dialog templates. That's what the "head" control palette item, shown here, is for.



You have a lot of freedom in designing your custom control. You can paint anything you want in its window (which is managed by the client application) and you can define any notification and inbound messages you need. You can use ClassWizard to map normal Windows messages in the control (WM\_LBUTTONDOWN, for example), but you must manually map the user-defined messages and manually map the notification messages in the parent window class.

## A Custom Control's Window Class

A dialog resource template specifies its custom controls by their symbolic window class names. Don't confuse the Win32 window class with the C++ class; the only similarity is the name. A window class is defined by a structure that contains the following:

* The name of the class
* A pointer to the *WndProc* function that receives messages sent to windows of the class
* Miscellaneous attributes, such as the background brush

The Win32 *RegisterClass* function copies the structure into process memory so that any function in the process can use the class to create a window. When the dialog window is initialized, Windows creates the custom control child windows from the window class names stored in the template.

Suppose now that the control's *WndProc* function is inside a DLL. When the DLL is initialized (by a call to *DllMain*), it can call *RegisterClass* for the control. Because the DLL is part of the process, the client program can create child windows of the custom control class. To summarize, the client knows the name string of a control window class and it uses that class name to construct the child window. All the code for the control, including the *WndProc* function, is inside the DLL. All that's necessary is that the client load the DLL prior to creating the child window.