LECTURE NOTES

ON

CS1305 – VISUAL PROGRAMMING

MR.M.S.DURAIRAJAN, M.E
ASST PROFESSOR
DEPT OF INFORMATION TECHNOLOGY
NPRCET

CS1305 – VISUAL PROGRAMMING
SYLLABUS

UNIT I WINDOWS PROGRAMMING

Windows environment − A simple windows program − Windows and messages − Creating the window − Displaying the window − Message loop − The Window procedure − Message processing − Text output − Painting and repainting − Introduction to GDI − Device context − Basic drawing − Child window controls.

UNIT II VISUAL C++ PROGRAMMING FUNDAMENTALS

Application framework − MFC library − Visual C++ components − Event handling − Mapping modes − Colors − Fonts − Modal and modeless dialog − Windows common controls – Bitmaps.

UNIT III THE DOCUMENT AND VIEW ARCHITECTURE

Menus − Keyboard accelerators − Rich edit control − Toolbars − Status bars − Reusable frame window base class − Separating document from its view – Reading and writing SDI and MDI Documents − Splitter window and multiple views − Creating DLLs − Dialog based applications.

UNIT IV ACTIVEX AND OBJECT LINKING AND EMBEDDING (OLE)

ActiveX Controls Vs Ordinary windows controls − Installing ActiveX Controls − Calendar Control − ActiveX control container programming − Create ActiveX control at runtime − Component Object Model (COM) − Containment and Aggregation Vs Inheritance − OLE Drag and Drop − OLE embedded component and containers − Sample applications.

UNIT V ADVANCED CONCEPTS

Database management with microsoft ODBC − Structured query language – MFC ODBC Classes − Sample database applications − Filter and Sort Strings – DAO Concepts − Displaying database records in scrolling view − Threading − VC++ Networking issues − WinSock − WinInet − Building a web client – Internet Information server − ISAPI server extension − Chat application − Playing and multimedia (sound and video) files.

TEXT BOOKS

1. Charles Petzold, “Windows Programming”, Microsoft Press, 1996.
2. David J. Kruglinski, George Shepherd and Scot Wingo, “Programming Visual C++”, Microsoft press, 1999.

REFERENCES

1. Steve Holtzner, “Visual C++ 6 Programming”, Wiley Dreamtech India Pvt. Ltd., 2003.
2. Mueller and John, “Visual C++ from the Ground up”, 2nd Edition, Tata McGraw Hill, 1999.
3. Bates and Tompkins, “Practical Visual C++”, Prentice Hall of India, 2002

UNIT 1

WINDOWS PROGRAMMING

The Windows Environment

A History of Windows

- Windows was announced by Microsoft Corporation in November 1983 (post-Lisa but pre-Macintosh) and was released two years later in November 1985
- Windows 2.0 was released in November 1987
- Windows 3.0 was introduced on May 22, 1990
- Microsoft Windows version 3.1 was released in April 1992
- Windows NT, introduced in July 1993
- Windows 95 was introduced in August 1995
- Windows 98 was released in June 1998

Aspects of Windows

- Both Windows 98 and Windows NT are 32-bit preemptive multitasking and multithreading graphical operating systems. Windows possesses a graphical user interface (GUI), sometimes also called a "visual interface" or "graphical windowing environment.“
- All GUIs make use of graphics on a bitmapped video display. Graphics provides better utilization of screen real estate, a visually rich environment for conveying information, and the possibility of a WYSIWYG (what you see is what you get) video display of graphics and formatted text prepared for a printed document.
- In earlier days, the video display was used solely to echo text that the user typed using the keyboard. In a graphical user interface, the video display itself becomes a source of user input. The video display shows various graphical objects in the form of icons and input devices such as buttons and scroll bars. Using the keyboard (or, more directly, a pointing device such as a mouse), the user can directly manipulate these objects on the screen. Graphics objects can be dragged, buttons can be pushed, and scroll bars can be scrolled

- The interaction between the user and a program thus becomes more intimate. Rather than the one-way cycle of information from the keyboard to the program to the video display, the user directly interacts with the objects on the display.
- Users no longer expect to spend long periods of time learning how to use the computer or mastering a new program. Windows helps because all applications have the same fundamental look and feel. The program occupies a window—usually a rectangular area on the screen.

- Each window is identified by a caption bar. Most program functions are initiated through the program's menus. A user can view the display of information too large to fit on a single screen by using scroll bars. Some menu items invoke dialog boxes, into which the user enters additional information.
- One dialog box in particular, that used to open a file, can be found in almost every large Windows program. This dialog box looks the same (or nearly the same) in all of these Windows programs, and it is almost always invoked from the same menu option.

- From the programmer's perspective, the consistent user interface results from using the routines built into Windows for constructing menus and dialog boxes. All menus have the same keyboard and mouse interface because Windows—rather than the application program—handles this job.

- To facilitate the use of multiple programs, and the exchange of information among them, Windows supports multitasking. Several Windows programs can be displayed and running at the same time. Each program occupies a window on the screen. The user can move the windows around on the screen, change their sizes, switch between different programs, and transfer data from one program to another.
- Earlier versions of Windows used a system of multitasking called "nonpreemptive." This meant that Windows did not use the system timer to slice processing time between the various programs running under the system. The programs themselves had to voluntarily give up control so that other programs could run. Under Windows NT and Windows 98, multitasking is preemptive and programs themselves can split into multiple threads of execution that seem to run concurrently.

- Programs running in Windows can share routines that are located in other files called "dynamic-link libraries." Windows includes a mechanism to link the program with the routines in the dynamic-link libraries at run time. Windows itself is basically a set of dynamic-link libraries.

- Windows is a graphical interface, and Windows programs can make full use of graphics and formatted text on both the video display and the printer. A graphical interface not only is more attractive in appearance but also can impart a high level of information to the user

- Programs written for Windows do not directly access the hardware of graphics display devices such as the screen and printer. Instead, Windows includes a graphics programming language (called the Graphics Device Interface, or GDI) that allows the easy display of graphics and formatted text. Windows virtualizes display hardware. A program written for Windows will run with any video board or any printer for which a Windows device driver is available. The program does not need to determine what type of device is attached to the system.

Dynamic Linking

- Windows provides a wealth of function calls that an application can take advantage of, mostly to implement its user interface and display text and graphics on the video display. These functions are implemented in dynamic-link libraries, or DLLs.
- When you run a Windows program, it interfaces to Windows through a process called "dynamic linking." A Windows .EXE file contains references to the various dynamic-link libraries it uses and the functions therein. When a Windows program is loaded into memory, the calls in the program are resolved to point to the entries of the DLL functions, which are also loaded into memory if not already there.
- When you link a Windows program to produce an executable file, you must link with special "import libraries" provided with your programming environment. These import libraries contain the dynamic-link library names and reference information for all the Windows function calls. The linker uses this information to construct the table in the .EXE file that Windows uses to resolve calls to Windows functions when loading the program.

Windows Programming Options

APIs and Memory Models

- To a programmer, an operating system is defined by its API. An API encompasses all the function calls that an application program can make of an operating system, as well as definitions of associated data types and structures.
- Windows NT and Windows 98 are both considered to support the Win32 API.

- Language Options

- Using C and the native APIs is not the only way to write programs for Windows 98. However, this approach offers you the best performance, the most power, and the greatest versatility in exploiting the features of Windows.
- Executables are relatively small and don't require external libraries to run (except for the Windows DLLs themselves, of course).
- Visual Basic or Borland Delphi Microsoft
- Microsoft Visual C++ with the Microsoft Foundation Class Library (MFC) - MFC encapsulates many of the messier aspects of Windows programming in a collection of C++ classes.

Your First Windows Program

```
/*------------------------------------------------------------
HelloMsg.c -- Displays "Hello, Windows 98!" in a message box
------------------------------------------------------------*/
#include <windows.h>
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,PSTR szCmdLine, int iCmdShow)
{
MessageBox (NULL, TEXT ("Hello, Windows 98!"), TEXT ("HelloMsg"), 0) ;
return 0 ;
}
```

The Header Files

HELLOMSG.C begins with a preprocessor directive that you'll find at the top of virtually every Windows program written in C:

#include <windows.h>

WINDOWS.H - master include file - includes other Windows header files, some of which also include other header files.
The most important and most basic of these header files are:

WINDEF.H **Basic type definitions.**
WINNT.H **Type definitions for Unicode support.**
WINBASE.H **Kernel functions.**
WINUSER.H **User interface functions.**
WINGDI.H **Graphics device interface functions.**

These header files define all the Windows data types, function calls, data structures, and constant identifiers.

Program Entry Point

Just as the entry point to a C program is the function *main*, the entry point to a Windows program is *WinMain*, which always appears like this:
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)

Hungarian Notation

b	BOOL
c or *ch*	char
clr	COLORREF
cx, cy	Horizontal or vertical distance
dw	DWORD
h	Handle
l	LONG
n	int
p	Pointer
sz	Zero-terminated string
w	WORD
wnd	CWnd
str	CString
m_	**class member variable**

- The *WinMain* function is declared as returning an *int*. The WINAPI identifier is defined in WINDEF.H with the statement:
- #define WINAPI __stdcall
- This statement specifies a calling convention that involves how machine code is generated to place function call arguments on the stack. Most Windows function calls are declared as WINAPI.

- The first parameter to *WinMain* is called an "instance handle." - simply a number that an application uses to identify the program.
- *The second parameter to WinMain* is always NULL (defined as 0).
- The third parameter to *WinMain* is the command line used to run the program.
- The fourth parameter to *WinMain* indicates how the program should be initially displayed—either normally or maximized to fill the window, or minimized to be displayed in the task list bar.

The *MessageBox* Function

- The *MessageBox* function is designed to display short messages. The little window that *MessageBox* displays is actually considered to be a dialog box, although not one with a lot of versatility.
- The first argument to *MessageBox* is normally a window handle.
- The second argument is the text string that appears in the body of the message box, and
- the third argument is the text string that appears in the caption bar of the message box.
- In HELLMSG.C, each of these text strings is enclosed in a TEXT macro.

- The fourth argument to *MessageBox* can be a combination of constants beginning with the prefix MB_ that are defined in WINUSER.H.
- You can pick one constant from the first set to indicate what buttons you wish to appear in the dialog box:

```
#define MB_OK                   0x00000000L
#define MB_OKCANCEL                 0x00000001L
#define MB_ABORTRETRYIGNORE         0x00000002L
#define MB_YESNOCANCEL          0x00000003L
```

#define MB_YESNO 0x00000004L
#define MB_RETRYCANCEL 0x00000005L
Windows Environment

Windows and Messages

Windows

- In Windows, the word "window" has a precise meaning. A window is a rectangular area on the screen that receives user input and displays output in the form of text and graphics.
- The *MessageBox* function creates a window, but it is a special-purpose window of limited flexibility. The message box window has a title bar with a close button, an optional icon, one or more lines of text, and up to four buttons. However, the icons and buttons must be chosen from a small collection that Windows provides for you.
- We can't display graphics in a message box, and we can't add a menu to a message box. For that we need to create our own windows.

An Architectural Overview

- The user sees windows as objects on the screen and interacts directly with them using the keyboard or the mouse. Interestingly enough, the programmer's perspective is analogous to the user's perspective.
- The window receives the user input in the form of "messages" to the window. A window also uses messages to communicate with other windows. Getting a good feel for messages is an important part of learning how to write programs for Windows.

- Every window that a program creates has an associated window procedure. This window procedure is a function that could be either in the program itself or in a dynamic-link library. Windows sends a message to a window by calling the window procedure. The window procedure does some processing based on the message and then returns control to Windows.
- More precisely, a window is always created based on a "window class." The window class identifies the window procedure that processes messages to the window. The use of a window class allows multiple windows to be based on the same window class and hence use the same window procedure. For example, all buttons in all Windows programs are based on the same window class. This window class is associated with a window procedure located in a Windows dynamic-link library that processes messages to all the button windows.

Creating the Window

- The window class defines general characteristics of a window, thus allowing the same window class to be used for creating many different windows.
- When you go ahead and create a window by calling *CreateWindow*, you specify more detailed information about the window.

- Why all the characteristics of a window can't be specified in one shot.
- Actually, dividing the information in this way is quite convenient. For example, all push-button windows are created based on the same window class. The window procedure associated with this window class is located inside Windows itself, and it is responsible for processing keyboard and mouse input to the push button and defining the button's visual appearance on the screen. All push buttons work the same way in this respect.

- But not all push buttons are the same. They almost certainly have different sizes, different locations on the screen, and different text strings. These latter characteristics are part of the window definition rather than the window class definition.

Creating the Window

```
hwnd = CreateWindow (szAppName,                 // window class name                TEXT ("The
Hello Program"),     // window caption          WS_OVERLAPPEDWINDOW,     // window style
                     CW_USEDEFAULT,             // initial x position
CW_USEDEFAULT,       // initial y position                                CW_USEDEFAULT,
     // initial x size                          CW_USEDEFAULT,           // initial y size
                     NULL,                      // parent window handle
NULL,                // window menu handle                               hInstance,
// program instance handle
NULL) ;              // creation parameters
```

- Creating the Window –Overlapped window
- It will have a title bar; a system menu button to the left of the title bar; a thick window-sizing border; and minimize, maximize, and close buttons to the right of the title bar. - standard style for windows,
- In WINUSER.H, this style is a combination of several bit flags:

```
#define WS_OVERLAPPEDWINDOW
(WS_OVERLAPPED      | \
WS_CAPTION      | \
WS_SYSMENU      | \                                   WS_THICKFRAME      | \
                                WS_MINIMIZEBOX     | \
        WS_MAXIMIZEBOX)
```

- By default, Windows positions successive newly created windows at stepped horizontal and vertical offsets from the upper left corner of the display.
- The *CreateWindow* call returns a handle to the created window. This handle is saved in the variable *hwnd*, which is defined to be of type HWND ("handle to a window").
- Every window in Windows has a handle. Your program uses the handle to refer to the window. Many Windows functions require *hwnd* as an argument so that Windows knows which window the function applies to. If a program creates many windows, each has a different handle. The handle to a window is one of the most important handles that a Windows program handles.

Displaying the Window

- After the *CreateWindow* call returns, the window has been created internally in Windows.
- What this means basically is that Windows has allocated a block of memory to hold all the information about the window that you specified in the *CreateWindow* call, plus some other information, all of which Windows can find later based on the window handle.
- However, the window does not yet appear on the video display. Two more calls are needed.

```
ShowWindow (hwnd, iCmdShow) ;
```

- The first argument is the handle to the window just created by *CreateWindow*.
- The second argument is the *iCmdShow* value passed as a parameter to *WinMain*. This determines how the window is to be initially displayed on the screen, whether it's normal, minimized, or maximized.
- The user probably selected a preference when adding the program to the Start menu. The value you receive from *WinMain* and pass to *ShowWindow* is SW_SHOWNORMAL if the window is displayed normally, SW_SHOWMAXIMIZED if the window is to be maximized, and SW_SHOWMINNOACTIVE if the window is just to be displayed in the taskbar.

- The *ShowWindow* function puts the window on the display. If the second argument to *ShowWindow* is SW_SHOWNORMAL, the client area of the window is erased with the background brush specified in the window class.
- UpdateWindow (hwnd) ;
- then causes the client area to be painted. It accomplishes this by sending the window procedure (the *WndProc* function in HELLOWIN.C) a WM_PAINT message.

The Message Loop

- After the *UpdateWindow* call, the window is fully visible on the video display. The program must now make itself ready to read keyboard and mouse input from the user.
- Windows maintains a "message queue" for each Windows program currently running under Windows. When an input event occurs, Windows translates the event into a "message" that it places in the program's message queue.
- A program retrieves these messages from the message queue by executing a block of code known as the "message loop":

```
while (GetMessage (&msg, NULL, 0, 0))
{
TranslateMessage (&msg) ;
DispatchMessage (&msg) ;
}
```

The Message Loop (2)

The *msg* variable is a structure of type MSG, which is defined in the WINUSER.H header file like this:

```
typedef struct tagMSG
{
HWND        hwnd ;
UINT        message ;
WPARAM   wParam ;
LPARAM   lParam ;
DWORD    time ;
POINT  pt ;
}
MSG, * PMSG ;
```

The POINT data type is yet another structure, defined in the WINDEF.H header file like this:

```
typedef struct tagPOINT
{
LONG  x ;
LONG  y ;
}
```

POINT, * PPOINT;

The Message Loop

TranslateMessage (&msg) ;

- **Passes the *msg* structure back to Windows for some keyboard translation.**
-

DispatchMessage (&msg) ;

- **Again passes the *msg* structure back to Windows.**

Windows then sends the message to the appropriate window procedure for processing

The Window Procedure

- The window class has been registered, the window has been created, the window has been displayed on the screen, and the program has entered a message loop to retrieve messages from the message queue.
- The real action occurs in the window procedure. The window procedure determines what the window displays in its client area and how the window responds to user input.

The Window Procedure

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

The four parameters to the window procedure are identical to the first four fields of the MSG structure

Processing the Messages

- Every message that a window procedure receives is identified by a number, which is the *message* parameter to the window procedure.
- The Windows header file WINUSER.H defines identifiers beginning with the prefix WM ("window message") for each type of message.
- Generally, Windows programmers use a *switch* and *case* construction to determine what message the window procedure is receiving and how to process it accordingly.
- When a window procedure processes a message, it should return 0 from the window procedure.
- All messages that a window procedure chooses not to process must be passed to a Windows function named *DefWindowProc*. The value returned from *DefWindowProc* must be returned from the window procedure.

In HELLOWIN, *WndProc* chooses to process only three messages: WM_CREATE, WM_PAINT, and WM_DESTROY. The window procedure is structured like this:

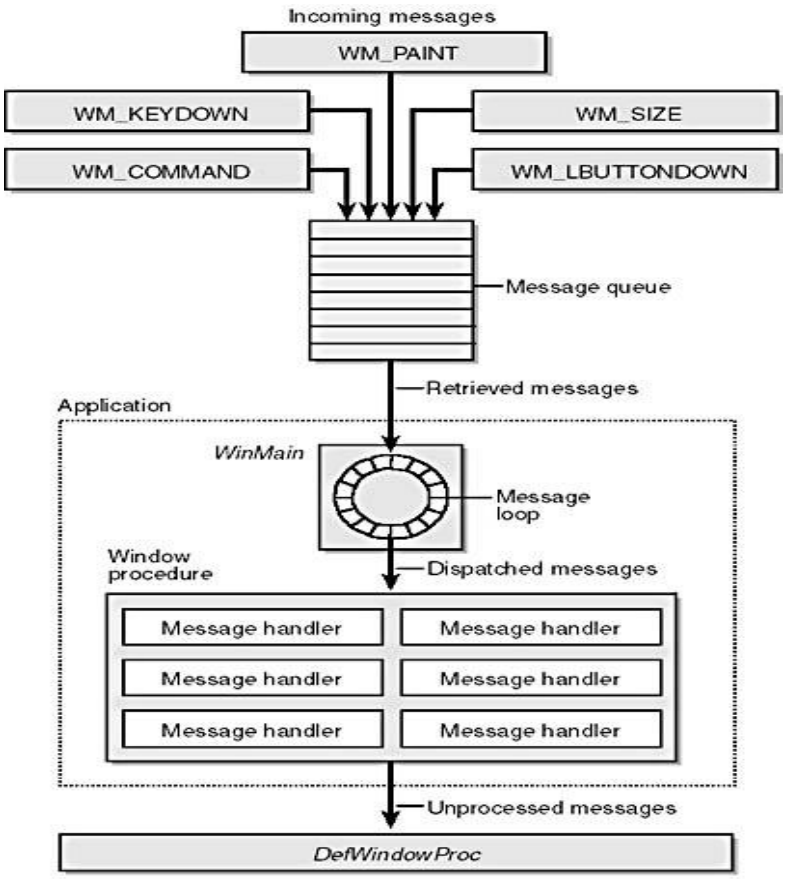
```
switch (iMsg)
{
case WM_CREATE :
[process WM_CREATE message]
```

```
return 0 ;
case WM_PAINT :
[process WM_PAINT message]
return 0 ;
case WM_DESTROY :
[process WM_DESTROY message]
return 0 ;
}
```

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;

- It is important to call *DefWindowProc* for default processing of all messages that your window procedure does not process. Otherwise behavior regarded as normal, such as being able to terminate the program, will not work.

WINDOWS PROGRAMMING MODEL



Text Output

Introduction

- Client area is the part of the window on which a program is free to draw and deliver visual information to the user.
- You can do almost anything you want with your program's client area—anything, that is, except assume that it will be a particular size or that the size will remain constant while your program is running.
- When a program displays text or graphics in its client area, it is often said to be "painting" its client area.
- Windows programs can assume little about the size of their client areas or even the size of text characters. Instead, they must use the facilities that Windows provides to obtain information about the environment in which the program runs.

Painting and Repainting

- In Windows, you can draw text and graphics only in the client area of your window, and you cannot be assured that what you put will remain there until your program specifically writes over it.
- For instance, the user may move another program's window on the screen so that it partially covers your application's window. Windows will not attempt to save the area of your window that the other program covers.When the program is moved away, Windows will request that your program repaint this portion of your client area.

- Windows is a message-driven system. Windows informs applications of various events by posting messages in the application's message queue or sending messages to the appropriate window procedure.
- Windows informs a window procedure that part of the window's client area needs painting by posting a WM_PAINT message.

The WM_PAINT Message

- Most Windows programs call the function *UpdateWindow* during initialization in *WinMain* shortly before entering the message loop.
- Windows takes this opportunity to send the window procedure its first WM_PAINT message. This message informs the window procedure that the client area must be painted.
- Thereafter, that window procedure should be ready at almost any time to process additional WM_PAINT messages and even to repaint the entire client area of the window if necessary.
- A window procedure receives a WM_PAINT message whenever one of the following events occurs:
- A previously hidden area of the window is brought into view when a user moves a window or uncovers a window.
- The user resizes the window (if the window class style has the CS_HREDRAW and CW_VREDRAW bits set).
- The program uses the *ScrollWindow* or *ScrollDC* function to scroll part of its client area.
- The program uses the *InvalidateRect* or *InvalidateRgn* function to explicitly generate a WM_PAINT message.

- Windows may sometimes post a WM_PAINT message when:
- Windows removes a dialog box or message box that was overlaying part of the window.
- A menu is pulled down and then released.
- A tool tip is displayed.
- In a few cases, Windows always saves the area of the display it overwrites and then restores it. This is the case whenever:
- The mouse cursor is moved across the client area.
- An icon is dragged across the client area.

- Your program should be structured so that it accumulates all the information necessary to paint the client area but paints only "on demand"—when Windows sends the window procedure a WM_PAINT message.
- If your program needs to update its client area at some other time, it can force Windows to generate this WM_PAINT message.

Valid and Invalid Rectangles

- Although a window procedure should be prepared to update the entire client area whenever it receives a WM_PAINT message, it often needs to update only a smaller area, most often a rectangular area within the client area. This is most obvious when a dialog box overlies part of the client area. Repainting is required only for the rectangular area uncovered when the dialog box is removed.
- That area is known as an "invalid region" or "update region." The presence of an invalid region in a client area is what prompts Windows to place a WM_PAINT message in the application's message queue. Your window procedure receives a WM_PAINT message only if part of your client area is invalid.

- Windows internally maintains a "paint information structure" for each window. This structure contains, among other information, the coordinates of the smallest rectangle that encompasses the invalid region. This is known as the "invalid rectangle.“
- If another region of the client area becomes invalid before the window procedure processes a pending WM_PAINT message, Windows calculates a new invalid region (and a new invalid rectangle) that encompasses both areas and stores this updated information in the paint information structure.
- Windows does not place multiple WM_PAINT messages in the message queue.

- A window procedure can invalidate a rectangle in its own client area by calling *InvalidateRect*.
- If the message queue already contains a WM_PAINT message, Windows calculates a new invalid rectangle. Otherwise, it places a WM_PAINT message in the message queue.
- A window procedure can obtain the coordinates of the invalid rectangle when it receives a WM_PAINT message It can also obtain these coordinates at any other time by calling *GetUpdateRect*.
- After the window procedure calls *BeginPaint* during the WM_PAINT message, the entire client area is validated. A program can also validate any rectangular area within the client area by calling the *ValidateRect* function. If this call has the effect of validating the entire invalid area, then any WM_PAINT message currently in the queue is removed.

Introduction

- The subsystem of Microsoft Windows responsible for displaying graphics on video displays and printers is known as the Graphics Device Interface (GDI).

- GDI is an extremely important part of Windows. Not only do the applications you write for Windows use GDI for the display of visual information, but Windows itself uses GDI for the visual display of user interface items such as menus, scroll bars, icons, and mouse cursors.

The Device Context

- When you want to draw on a graphics output device such as the screen or printer, you must first obtain a handle to a device context (or DC).
- In giving your program this handle, Windows is giving you permission to use the device.
- You then include the handle as an argument to the GDI functions to identify to Windows the device on which you wish to draw.
- The device context contains many "attributes" that determine how the GDI functions work on the device.
- These attributes allow GDI functions to have just a few arguments, such as starting coordinates. The GDI functions do not need arguments for everything else that Windows needs to display the object on the device.

- For example, when you call *TextOut*, you need specify in the function only the device context handle, the starting coordinates, the text, and the length of the text.
- You don't need to specify the font, the color of the text, the color of the background behind the text, or the intercharacter spacing.
- These are all attributes that are part of the device context.
- When you want to change one of these attributes, you call a function that does so. Subsequent *TextOut* calls to that device context use the new attribute.

Getting a Device Context Handle

- Windows provides several methods for obtaining a device context handle. If you obtain a video display device context handle while processing a message, you should release it before exiting the window procedure. After you release the handle, it is no longer valid.
- The most common method for obtaining a device context handle and then releasing it involves using the *BeginPaint* and *EndPaint* calls when processing the WM_PAINT message:

```
hdc = BeginPaint (hwnd, &ps) ;
[other program lines]
EndPaint (hwnd, &ps) ;
```

Getting a Device Context Handle

Windows programs can also obtain a handle to a device context while processing messages other than WM_PAINT:

```
hdc = GetDC (hwnd) ;
[other program lines]
ReleaseDC (hwnd, hdc) ;
```

Getting a Device Context Handle

The *BeginPaint*, *GetDC*, and *GetWindowDC* calls obtain a device context associated with a particular window on the video display.

A much more general function for obtaining a handle to a device context is *CreateDC*:

```
hdc = CreateDC (pszDriver, pszDevice, pszOutput, pData) ;
```

[other program lines]
DeleteDC (hdc) ;

Getting Device Context Information

- A device context usually refers to a physical display device such as a video display or a printer.
- Often, you need to obtain information about this device, including the size of the display, in terms of both pixels and physical dimensions, and its color capabilities.
- You can get this information by calling the *GetDeviceCap* ("get device capabilities") function:
- iValue = GetDeviceCaps (hdc, iIndex) ;

- The *iIndex* argument is one of 29 identifiers defined in the WINGDI.H header file.
- For example, the *iIndex* value of HORZRES causes *GetDeviceCaps* to return the width of the device in pixels; a VERTRES argument returns the height of the device in pixels.
- If *hdc* is a handle to a screen device context, that's the same information you can get from *GetSystemMetrics*.
- If *hdc* is a handle to a printer device context, *GetDeviceCaps* returns the height and width of the printer display area in pixels.
- You can also use *GetDeviceCaps* to determine the device's capabilities of processing various types of graphics.
- *DEVCAPS1 display for a 256-color, 640-by-480 VGA.*

The Size of the Device

- The *GetDeviceCaps* function helps you obtain information regarding the physical size of the output device, be it the video display or printer.
- Within a Windows program you can use the *GetDeviceCaps* function to obtain the assumed resolution in dots per inch that the user selected in the Display applet of the Control Panel

Finding Out About Color

- A video display capable of displaying only black pixels and white pixels requires only one bit of memory per pixel. Color displays require multiple bits per pixels. The more bits, the more colors; or more specifically, the number of unique simultaneous colors is equal to 2 to the number of bits per pixel.
- iBitsPixel = GetDeviceCaps (hdc, BITSPIXEL) ;
- iColors = GetDeviceCaps (hdc, NUMCOLORS) ;

The Device Context Attributes

Saving Device Contexts

- Normally when you call *GetDC* or *BeginPaint*, Windows gives you a device context with default values for all the attributes.
- Any changes you make to the attributes are lost when the device context is released with the *ReleaseDC* or *EndPaint* call.
- If your program needs to use non-default device context attributes, you'll have to initialize the device context every time you obtain a new device context handle:

Saving Device Contexts

- Although this approach is generally satisfactory, you might prefer that changes you make to the attributes be saved when you release the device context so that they will be in effect the next time you call *GetDC* or *BeginPaint*.
- You can accomplish this by including the CS_OWNDC flag as part of the window class style when you register the window class:
- wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC ;

Saving Device Contexts

- Now each window that you create based on this window class will have its own private device context that continues to exist when the window is destroyed.
- When you use the CS_OWNDC style, you need to initialize the device context attributes only once, perhaps while processing the WM_CREATE message:
- case WM_CREATE: hdc = GetDC (hwnd) ;
- *[initialize device context attributes]*
- ReleaseDC (hwnd, hdc) ;
- The attributes continue to be valid until you change them.

Saving Device Contexts

- In some cases you might want to change certain device context attributes, do some painting using the changed attributes, and then revert to the original device context. To simplify this process, you save the state of a device context by calling
- idSaved = SaveDC (hdc) ;
- Now you can change some attributes. When you want to return to the device context as it existed before the *SaveDC* call, you use
- RestoreDC (hdc, idSaved) ;
- You can call *SaveDC* any number of times before you call *RestoreDC*.

Child Window Controls

Introduction

- The child window processes mouse and keyboard messages and notifies the parent window when the child window's state has changed.
- In this way, the child window becomes a high-level input device for the parent window.
- It encapsulates a specific functionality with regard to its graphical appearance on the screen, its response to user input, and its method of notifying another window when an important input event has occurred.

Child Window Controls

- Push Buttons
- Check Boxes
- Radio Buttons
- Group Boxes
- Controls and Colors
- The Scroll Bar Class
- The Edit Class

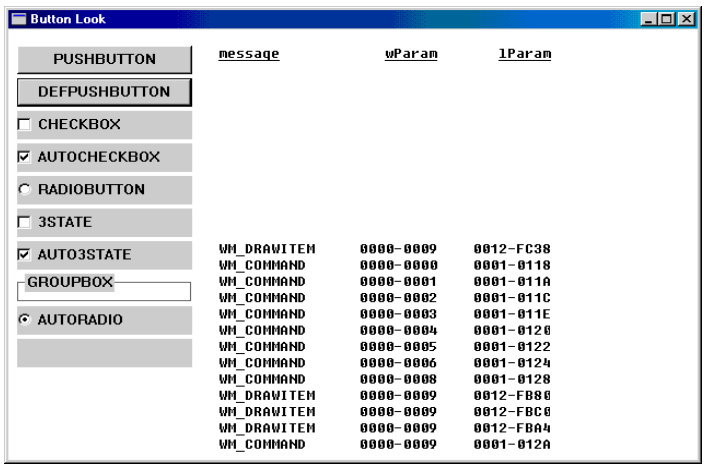
- The Listbox Class

Creating the Child Windows

Class name	TEXT ("button")
Window text	button[i].szText
Window style	WS_CHILD ¦ WS_VISIBLE ¦ button[i].iStyle
x position	cxChar
y position	cyChar * (1 + 2 * i)
Width	20 * xChar
Height	7 * yChar / 4
Parent window	hwnd
Child window	ID(HMENU) I
Instance handle	((LPCREATESTRUCT) lParam) -> hInstance
Extra parameters	NULL

Push Buttons

- A push button is a rectangle enclosing text specified in the window text parameter of the *CreateWindow* call. The rectangle takes up the full height and width of the dimensions given in the *CreateWindow* or *MoveWindow* call. The text is centered within the rectangle.
- Push-button controls are used mostly to trigger an immediate action without retaining any type of on/off indication.
- The two types of push-button controls have window styles called BS_PUSHBUTTON and BS_DEFPUSHBUTTON.



- When used to design dialog boxes, BS_PUSHBUTTON controls and BS_DEFPUSHBUTTON controls function differently from one another. When used as child window controls, however, the

two types of push buttons function the same way, although BS_DEFPUSHBUTTON has a heavier outline.

- A push button looks best when its height is 7/4 times the height of a text character, which is what BTNLOOK uses. The push button's width must accommodate at least the width of the text, plus two additional characters.
- When the mouse cursor is inside the push button, pressing the mouse button causes the button to repaint itself using 3D-style shading to appear as if it's been depressed. Releasing the mouse button restores the original appearance and sends a WM_COMMAND message to the parent window with the notification code BN_CLICKED.
- As with the other button types, when a push button has the input focus, a dashed line surrounds the text and pressing and releasing the Spacebar has the same effect as pressing and releasing the mouse button.

- You can simulate a push-button flash by sending the window a BM_SETSTATE message.
- This causes the button to be depressed:

```
SendMessage (hwndButton, BM_SETSTATE, 1, 0) ;
This call causes the button to return to normal:
SendMessage (hwndButton, BM_SETSTATE, 0, 0) ;
```

Check Boxes

- A check box is a square box with text; the text usually appears to the right of the check box.
- If you include the BS_LEFTTEXT style when creating the button, the text appears to the left; you'll probably want to combine this style with BS_RIGHT to right-justify the text.
- Check boxes are usually incorporated in an application to allow a user to select options.
- The check box commonly functions as a toggle switch: clicking the box once causes a check mark to appear; clicking again toggles the check mark off.

- The two most common styles for a check box are BS_CHECKBOX and BS_AUTOCHECKBOX.
- When you use the BS_CHECKBOX style, you must set the check mark yourself by sending the control a BM_SETCHECK message. The *wParam* parameter is set to 1 to create a check mark and to 0 to remove it.
- You can obtain the current check state of the box by sending the control a BM_GETCHECK message. You might use code like this to toggle the X mark when processing a WM_COMMAND message from the control:
- SendMessage ((HWND) lParam, BM_SETCHECK, (WPARAM)
- !SendMessage ((HWND) lParam, BM_GETCHECK, 0, 0), 0) ;

- The other two check box styles are BS_3STATE and BS_AUTO3STATE. As their names indicate, these styles can display a third state as well—a gray color within the check box—which occurs when you send the control a WM_SETCHECK message with *wParam* equal to 2. The gray color indicates to the user that the selection is indeterminate or irrelevant.
- The check box is aligned with the rectangle's left edge and is centered within the top and bottom dimensions of the rectangle that were specified during the *CreateWindow* call. Clicking anywhere within the rectangle causes a WM_COMMAND message to be sent to the parent. The minimum height for a check box is one character height. The minimum width is the number of characters in the text, plus two.

Radio Buttons

- A radio button is named after the row of buttons that were once quite common on car radios. Each button on a car radio is set for a different radio station, and only one button can be pressed at a time. In dialog boxes, groups of radio buttons are conventionally used to indicate mutually exclusive options. Unlike check boxes, radio buttons do not work as toggles—that is, when you click a radio button a second time, its state remains unchanged.
- The radio button looks very much like a check box except that it contains a little circle rather than a box. A heavy dot within the circle indicates that the radio button has been checked. The radio button has the window style BS_RADIOBUTTON or BS_AUTORADIOBUTTON, but the latter is used only in dialog boxes.

Group Boxes

- The group box, which has the BS_GROUPBOX style, is an oddity in the button class.
- It neither processes mouse or keyboard input nor sends WM_COMMAND messages to its parent.
- The group box is a rectangular outline with its window text at the top. Group boxes are often used to enclose other button controls.

The Scroll Bar Class

- You add window scroll bars to a window by including the identifier WS_VSCROLL or WS_HSCROLL or both in the window style when creating the window.
- You create child window scroll bar controls by using the predefined window class "scrollbar" and one of the two scroll bar styles SBS_VERT and SBS_HORZ.

- Unlike the button controls, scroll bar controls do not send WM_COMMAND messages to the parent window.
- Instead, they send WM_VSCROLL and WM_HSCROLL messages, just like window scroll bars.
- When processing the scroll bar messages, you can differentiate between window scroll bars and scroll bar controls by the *lParam* parameter. It will be 0 for window scroll bars and the scroll bar window handle for scroll bar controls. The high and low words of the *wParam* parameter have the same meaning for window scroll bars and scroll bar controls.
- Although window scroll bars have a fixed width, Windows uses the full rectangle dimensions given in the *CreateWindow* call to size the scroll bar controls.
- You can make long, thin scroll bar controls or short, pudgy scroll bar controls.

- You can set the range and position of a scroll bar control with the same calls used for window scroll bars:
- SetScrollRange (hwndScroll, SB_CTL, iMin, iMax, bRedraw) ;
- SetScrollPos (hwndScroll, SB_CTL, iPos, bRedraw) ;
- SetScrollInfo (hwndScroll, SB_CTL, &si, bRedraw) ;
- The difference is that window scroll bars use a handle to the main window as the first parameter and SB_VERT or SB_HORZ as the second parameter.

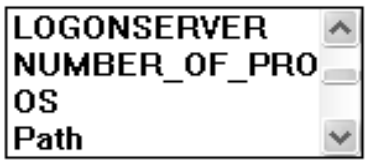
The Edit Class

- When you create a child window using the class name "edit," you define a rectangle based on the *x* position, *y* position, width, and height parameters of the *CreateWindow* call.
- This rectangle contains editable text. When the child window control has the input focus, you can type text, move the cursor, select portions of text using either the mouse or the Shift key and a cursor key, delete selected text to the clipboard by pressing Ctrl-X, copy text by pressing Ctrl-C, and insert text from the clipboard by pressing Ctrl-V.
- One of the simplest uses of edit controls is for single-line entry fields. But edit controls are not limited to single lines, to use menus, dialog boxes (to load and save files), and printing.

- Create an edit control using "edit" as the window class in the *CreateWindow* call. The window style is WS_CHILD, plus several options. As in static child window controls, the text in edit controls can be left-justified, right-justified, or centered. You specify this formatting with the window styles ES_LEFT, ES_RIGHT, and ES_CENTER.
- By default, an edit control has a single line. You can create a multiline edit control with the window style ES_MULTILINE.
- To create an edit control that automatically scrolls horizontally, you use the style ES_AUTOHSCROLL. For a multiline edit control, text wordwraps unless you use the ES_AUTOHSCROLL style, in which case you must press the Enter key to start a new line. You can also include vertical scrolling in a multiline edit control by using the style ES_AUTOVSCROLL.
- To add scroll bars to the edit control.-use the same window style identifiers as for nonchild windows: WS_HSCROLL and WS_VSCROLL.
- By default, an edit control does not have a border. You can add one by using the style WS_BORDER.

- When you select text in an edit control, Windows displays it in reverse video. When the edit control loses the input focus, however, the selected text is no longer highlighted. If you want the selection to be highlighted even when the edit control does not have the input focus, you can use the style ES_NOHIDESEL.
- Style given in the *CreateWindow* call:
- WS_CHILD ¦ WS_VISIBLE ¦ WS_HSCROLL ¦ WS_VSCROLL ¦ WS_BORDER ¦ ES_LEFT ¦ ES_MULTILINE ¦ ES_AUTOHSCROLL ¦ ES_AUTOVSCROLL

The Listbox Class



- A list box is a collection of text strings displayed as a scrollable columnar list within a rectangle. A program can add or remove strings in the list by sending messages to the list box window procedure. The list box control sends WM_COMMAND messages to its parent window when an item in the list is selected. The parent window can then determine which item has been selected.
- A list box can be either single selection or multiple selection. The latter allows the user to select more than one item from the list box. When a list box has the input focus, it displays a dashed line surrounding an item in the list box. This cursor does not indicate the selected item in the list box. The selected item is indicated by highlighting, which displays the item in reverse video.

- In a single-selection list box, the user can select the item that the cursor is positioned on by pressing the Spacebar. The arrow keys move both the cursor and the current selection and can

scroll the contents of the list box. The Page Up and Page Down keys also scroll the list box by moving the cursor but not the selection. Pressing a letter key moves the cursor and the selection to the first (or next) item that begins with that letter. An item can also be selected by clicking or double-clicking the mouse on the item.

- In a multiple-selection list box, the Spacebar toggles the selection state of the item where the cursor is positioned. (If the item is already selected, it is deselected.) The arrow keys deselect all previously selected items and move the cursor and selection, just as in single-selection list boxes. However, the Ctrl key and the arrow keys can move the cursor without moving the selection. The Shift key and arrow keys can extend a selection.

- Clicking or double-clicking an item in a multiple-selection list box deselects all previously selected items and selects the clicked item. However, clicking an item while pressing the Shift key toggles the selection state of the item without changing the selection state of any other item.

PART – A (2 MARKS)

1. List out the aspects of Windows
2. Define Dynamic Link Libraries
3. List out the types of DLL which is implemented in Windows.
4. Define Window Procedure
5. Define Message Queue & Message Loop
6. Define handle
7. Define Hungarian Notation
8. What are the events used to generate a WM_PAINT message?
9. Define Invalid region
10. Define Invalid rectangle
11. Define Device Context
12. List out the aspects of GDI
13. Define System font
14. Define Dithering
15. List out the GDI Primitives
16. List out the pen styles
17. Define Mapping Modes
18. Define Viewport and window
19. Define Raster Operation
20. Define child window control

PART – B

1. Explain in detail about various versions of Windows Operating System(16)
2. Explain briefly about,
a. How to create a window (6)
b. Displaying the window (4)
c. Processing the message (6)
3. a. Describe the functions of Message Loop (8)
b. Explain in detail about the Windows Message Structure and Windows Procedure. (8)
4. a. How does the WM_PAINT message is processed? (10)
b. What is WM_DESTROY message? How the program is terminated? (6)
5. a. Define DC. (2)
b. What are the methods available to get the DC and various types of DC Handle? (14)
6. Explain Windows Graphics Device Interface in detail (16)
7. a. Write a note on Hungarian Notation in Windows Programming (6)
b. Write a program to display a message in the center of a window (10)
8. a. Explain the methods of getting device context handle (8).

UNIT II
MICROSOFT FOUNDATION CLASS

The Microsoft Foundation Class Library Application Framework

- **Application framework**

"An integrated collection of object-oriented software components that offers all that's needed for a generic application."

- **An Application Framework vs. a Class Library**

- An application framework is a superset of a class library.
- An ordinary library is an isolated set of classes designed to be incorporated into any program, but an application framework defines the structure of the program itself.

Why Use the Application Framework?

- **The MFC library is the C++ Microsoft Windows API.**
- **Application framework applications use a standard structure.**
- **Application framework applications are small and fast.**
- **The Visual C++ tools reduce coding drudgery**
- **The MFC library application framework is feature rich**
- **An Application Framework Example**

source code for the header and implementation files for our MYAPPapplication.

- **MyApp.h header file for the MYAPP application:**

```
// application class class
CMyApp : public CWinApp
 {
  public:
      virtual BOOL InitInstance();
 };
// frame window class class
CMyFrame : public CFrameWnd
{
 public:
   CMyFrame();
 protected:
   // "afx_msg" indicates that the next two functions are part
   // of the MFC library message dispatch system afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP()
 };
```

MyApp.cpp - implementation file for the MYAPP application:

```
#include <afxwin.h> // MFC library header file declares base classes
#include "myapp.h"
CMyApp theApp; // the one and only CMyApp object
BOOL CMyApp::InitInstance()
{ m_pMainWnd = new CMyFrame();
 m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
```

```
return TRUE;
 }
BEGIN_MESSAGE_MAP(CMyFrame, CFrameWnd) ON_WM_LBUTTONDOWN()
    ON_WM_PAINT()
END_MESSAGE_MAP()

CMyFrame::CMyFrame()
{
Create(NULL, "MYAPP Application");
}
void CMyFrame::OnLButtonDown(UINT nFlags, CPoint point)
{ TRACE("Entering CMyFrame::OnLButtonDown - %lx, %d, %d\n", (long) nFlags, point.x, point.y);
}
void CMyFrame::OnPaint()
{ CPaintDC dc(this);
dc.TextOut(0, 0, "Hello, world!");
}
```

The program elements:

The *WinMain* function

Windows requires your application to have a *WinMain* function. You don't see *WinMain* here because it's hidden inside the application framework.

The *CMyApp* class

An object of class *CMyApp* represents an application. The program defines a single global *CMyApp* object, *theApp*. The *CWinApp* base class determines most of *theApp*'s behavior.

Application startup

When the user starts the application, Windows calls the application framework's built-in *WinMain* function, and *WinMain* looks for your globally constructed application object of a class derived from *CWinApp*.

In a C++ program global objects are constructed before the main program is executed.

The *CMyApp::InitInstance* member function

When the *WinMain* function finds the application object, it calls the virtual *InitInstance* member function, which makes the calls needed to construct and display the application's main frame window. You must override *InitInstance* in your derived application class because the *CWinApp* base class doesn't know what kind of main frame window you want.

The *CWinApp::Run* member function

The *Run* function is hidden in the base class, but it dispatches the application's messages to its windows, thus keeping the application running. *WinMain* calls *Run* after it calls *InitInstance*.

The *CMyFrame* class

An object of class *CMyFrame* represents the application's main frame window. When the constructor calls the *Create* member function of the base class *CFrameWnd*, Windows creates the actual window structure and the application framework links it to the C++ object. The *ShowWindow* and *UpdateWindow* functions, also member functions of the base class, must be called in order to display the window.

- **The program elements:**

The *CMyFrame::OnLButtonDown* function

MFC library's message-handling capability.

The function invokes the MFC library *TRACE* macro to display a message in the debugging window.

The *CMyFrame::OnPaint* function

- The application framework calls this important mapped member function of class *CMyFrame* every time it's necessary to repaint the window: at the start of the program, when the user resizes the window, and when all or part of the window is newly exposed.
- The *CPaintDC* statement relates to the Graphics Device Interface (GDI) and is explained in later chapters. The *TextOut* function displays "Hello, world!"

Application shutdown

- The user shuts down the application by closing the main frame window.
- This action initiates a sequence of events, which ends with the destruction of the *CMyFrame* object, the exit from *Run*, the exit from *WinMain*, and the destruction of the *CMyApp* object.

MFC Library Message Mapping

- The MFC library application framework doesn't use virtual functions for Windows messages. Instead, it uses macros to "map" specified messages to derived class member functions

Why the rejection of virtual functions?

- What about message handlers for menu command messages and messages from button clicks?
- An MFC message handler requires a function prototype, a function body, and an entry (macro invocation) in the message map.

```
BEGIN_MESSAGE_MAP(CMyFrame, CFrameWnd)
    ON_WM_LBUTTONDOWN()
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

- **Documents and Views**
- Typically, MFC application will contain application and frame classes plus two other classes that represent the "document" and the "view."
- This document-view architecture is the core of the application framework
- The document-view architecture separates data from the user's view of the data. One obvious benefit is multiple views of the same data.

The Visual C++ Components

- Microsoft Visual C++ is two complete Windows application development systems in one product.
- You can develop C-language Windows programs using only the Win32 API.
- You can use many Visual C++ tools, including the resource editors, to make low-level Win32 programming easier.
- **Components:**
  - The Project
  - The Resource Editors—Workspace ResourceView
  - The C/C++ Compiler

- The Source Code Editor
- The Resource Compiler
- The Linker
- The Debugger
- AppWizard
- Classwizard

What is a project?

- A project is a collection of interrelated source files that are compiled and linked to make up an executable Windows-based program or a DLL.
- Source files for each project are generally stored in a separate subdirectory.
- A project depends on many files outside the project subdirectory too, such as include files and library files.

A makefile stores compiler and linker options and expresses all the interrelationships among source files. A make program reads the makefile and then invokes the compiler, assembler, resource compiler, and linker to produce the final output, which is generally an executable file.

In a Visual C++ 6.0 project, there is no makefile (with an MAK extension) unless you tell the system to export one.

- A text-format project file (with a DSP extension) serves the same purpose.
- A separate text-format workspace file (with a DSW extension) has an entry for each project in the workspace.
- It's possible to have multiple projects in a workspace, but all the

Examples in this book have just one project per workspace.

- To work on an existing project, you tell Visual C++ to open the DSW file and then you can edit and build the project.
- VC++ Project Files

Visual C++ creates some intermediate files too

File Extension	**Description**
APS	Supports ResourceView
BSC	Browser information file
CLW	Supports ClassWizard
DEP	Dependency file
DSP	Project file
*DSW	Workspace file
*MAK	External makefile
NCB	Supports ClassView
OPT	Holds workspace configuration
PLG	Builds log file

* Do not delete or edit in a text editor.

The Resource Editors— Workspace ResourceView

- Each project usually has one text-format resource script (RC) file that describes the project's menu, dialog, string, and accelerator resources.
- The RC file also has *#include* statements to bring in resources from other subdirectories.
- These resources include project-specific items, such as bitmap (BMP) and icon (ICO) files, and resources common to all Visual C++ programs, such as error message strings.
- Editing the RC file outside the resource editors is not recommended.
- The resource editors can also process EXE and DLL files, so you can use the clipboard to "steal" resources, such as bitmaps and icons, from other Windows applications.
- **The C/C++ Compiler**
- The Visual C++ compiler can process both C source code and C++ source code.
- It determines the language by looking at the source code's filename extension.
- A C extension indicates C source code, and CPP or CXX indicates C++ source code.
- The compiler is compliant with all ANSI standards, including the latest recommendations of a working group on C++ libraries, and has additional Microsoft extensions.
- Templates, exceptions, and runtime type identification (RTTI) are fully supported in Visual C++ version 6.0.
- The C++ Standard Template Library (STL) is also included, although it is not integrated into the MFC library.

The Other Components

- The Source Code Editor

  Visual C++ 6.0 includes a sophisticated source code editor that supports many features such as dynamic syntax coloring, auto-tabbing, keyboard bindings

- The Resource Compiler

The Visual C++ resource compiler reads an ASCII resource script (RC) file from the resource editors and writes a binary RES file for the linker.

- The Linker

The linker reads the OBJ and RES files produced by the C/C++ compiler and the resource compiler, and it accesses LIB files for MFC code, runtime library code, and Windows code. It then writes the project's EXE file.

The Debugger

- The Visual C++ debugger has been steadily improving, but it doesn't actually fix the bugs yet. The debugger works closely with Visual C++ to ensure that breakpoints are saved on disk.

AppWizard

- **AppWizard is a code generator that creates a working skeleton of a Windows application with features, class names, and source code filenames that you specify through dialog boxes.**
- **AppWizard code is minimalist code; the functionality is inside the application framework base classes.**
- **AppWizard gets you started quickly with a new application.**

- **ClassWizard**
  - **ClassWizard is a program (implemented as a DLL) that's accessible from Visual C++'s View menu.**
  - **ClassWizard takes the drudgery out of maintaining Visual C++ class code.**
  - **Need a new class, a new virtual function, or a new message-handler function?**
    - ✓ **ClassWizard writes the prototypes, the function bodies, and (if necessary) the code to link the Windows message to the function.**
    - ✓ **ClassWizard can update class code that you write, so you avoid the maintenance problems common to ordinary code generators.**

Basic Event Handling, Mapping Modes, and a Scrolling View

The Message Handler:

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
 // event processing code here
 }
```

The Message Map:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
ON_WM_LBUTTONDOWN()
                    // entry specifically for OnLButtonDown
        // other message map entries
END_MESSAGE_MAP()
```

Finally, your class header file needs the statement

```
DECLARE_MESSAGE_MAP()
```

Invalid Rectangle Theory

InvalidateRect triggers a Windows **WM_PAINT** message, which is mapped in the *CView* class to call to the virtual *OnDraw* function.

If necessary, ***OnDraw*** can access the "invalid rectangle" parameter that was passed to ***InvalidateRect.***

Your ***OnDraw*** function could call the ***CDC*** member function ***GetClipBox*** to determine the invalid rectangle, and then it could avoid drawing objects outside it.

OnDraw is being called not only in response to your ***InvalidateRect*** call but also when the user resizes or exposes the window.

Thus, ***OnDraw*** is responsible for all drawing in a window, and it has to adapt to whatever invalid rectangle it gets.

The Window's Client Area

A window has a rectangular client area that excludes the border, caption bar, menu bar, and any toolbars. The *CWnd* member function *GetClientRect* supplies you with the client-area dimensions.
Normally, you're not allowed to draw outside the client area, and most mouse messages are received only when the mouse cursor is in the client area.

***CRect*, *CPoint*, and *CSize* Arithmetic**

The *CRect*, *CPoint*, and *CSize* classes are derived from the Windows *RECT*, *POINT*, and *SIZE* structures, and thus they inherit public integer data members as follows:

CRect	*left, top, right, bottom*
CPoint	*x, y*
CSize	*cx, cy*

- **Dialog : Using Appwizard and Classwizard**
- **The Modal Dialog and Windows Common Controls**

- The two kinds of dialogs are modal and modeless.
- The *CDialog* base class supports both modal and modeless dialogs

Modal Dialog Box:

The user cannot work elsewhere in the same application (more correctly, in the same user interface thread) until the dialog is closed. Example: Open File dialog

Modeless Dialog

The user can work in another window in the application while the dialog remains on the screen

Example: Microsoft Word's Find and Replace dialog is a good example of a modeless dialog; you can edit your document while the dialog is open.

Controls.

A dialog contains a number of elements called controls. Dialog controls include edit controls, buttons, list boxes, combo boxes, static text, tree views, progress indicators, sliders, and so forth.

- **Programming a Modal Dialog**

1. Use the dialog editor to create a dialog resource that contains various controls.
2. -The dialog editor updates the project's resource script (RC) file to include your new dialog resource, and it updates the project's resource.h file with corresponding *#define* constants.
3. Use ClassWizard to create a dialog class that is derived from *CDialog* and attached to the resource created in step 1.
4. -ClassWizard adds the associated code and header file to the Microsoft Visual C++ project.
5. Use ClassWizard to add data members, exchange functions, and validation functions to the dialog class.
6. Use ClassWizard to add message handlers for the dialog's buttons and other event-generating controls.
7. Write the code for special control initialization (in *OnInitDialog*) and for the message handlers. Be sure the *CDialog* virtual member function *OnOK* is called when the user closes the dialog (unless the user cancels the dialog). (Note: *OnOK* is called by default.)
8. Write the code in your view class to activate the dialog. This code consists of a call to your dialog class's constructor followed by a call to the *DoModal* dialog class member function. *DoModal* returns only when the user exits the dialog window.

In the CPP file, the constructor implementation looks like this:

```
CMyDialog::CMyDialog(CWnd* pParent /*=NULL*/) : CDialog(CMyDialog::IDD, pParent)
{
// initialization code here
}
```

The use of *enum IDD* decouples the CPP file from the resource IDs that are defined in the project's resource.h

- **The Windows Common Dialogs**
- Windows provides a group of standard user interface dialogs, and these are supported by the MFC library classes.
- All the common dialog classes are derived from a common base class, *CCommonDialog*.

Class	**Purpose**
CColorDialog	Allows the user to select or create a color
CFileDialog	Allows the user to open or save a file
CFindReplaceDialog	Allows the user to substitute one string for another
CPageSetupDialog	Allows the user to input page measurement parameters
CFontDialog	Allows the user to select a font from a list of available fonts
CPrintDialog	Allows the user to set up the printer and print a document

- ***Using the* CFileDialog *Class Directly***
- The following code opens a file that the user has selected through the dialog:

```
CFileDialog dlg(TRUE, "bmp", "*.bmp");
if (dlg.DoModal() == IDOK) {
CFile file;
VERIFY(file.Open(dlg.GetPathName(), CFile::modeRead));
}
```

- The first constructor parameter (*TRUE*) specifies that this object is a "File Open" dialog instead of a "File Save" dialog.
- The default file extension is *bmp*, and **.bmp* appears first in the filename edit box. The *CFileDialog::GetPathName* function returns a *CString* object that contains the full pathname of the selected file.

- BITMAPS
- Windows bitmaps are arrays of bits mapped to display pixels.
- There are two kinds of Windows bitmaps: GDI bitmaps and DIBs.
- GDI bitmap objects are represented by the Microsoft Foundation Class (MFC) Library version 6.0

Color Bitmaps and Monochrome Bitmaps

- Many color bitmaps are 16-color. A standard VGA board has four contiguous color planes, with 1 corresponding bit from each plane combining to represent a pixel.
- The 4-bit color values are set when the bitmap is created. With a standard VGA board, bitmap colors are limited to the standard 16 colors. Windows does not use dithered colors in bitmaps.
- A monochrome bitmap has only one plane. Each pixel is represented by a single bit that is either off (0) or on (1). The *CDC::SetTextColor* function sets the "off" display color, and *SetBkColor* sets the "on" color.
- You can specify these pure colors individually with the Windows *RGB* macro.
- Code to load a Bitmap

```
void OnPaint()
```

```
    {
CBitmap mybm;
     CPaintDC d(this);
     mybm.LoadBitmap(IDB_BITMAP1);
     CBrush mybrush;
     mybrush.CreatePatternBrush(&mybm);
d.SelectObject(&mybrush);
     d.Rectangle(100,100,300,300);
    }
```

- GDI Bitmaps and Device-Independent Bitmaps

GDI Bitmaps

- There are two kinds of Windows bitmaps: GDI bitmaps and DIBs.
- GDI bitmap objects are represented by the Microsoft Foundation Class (MFC) Library version 6.0 *CBitmap* class.
- The GDI bitmap object has an associated Windows data structure, maintained inside the Windows GDI module, that is device-dependent.
- Your program can get a copy of the bitmap data, but the bit arrangement depends on the display hardware.
- GDI bitmaps can be freely transferred among programs on a single computer, but because of their device dependency, transferring bitmaps by disk or modem doesn't make sense.
- Device-Independent Bitmaps

Device-Independent Bitmaps

- DIBs offer many programming advantages over GDI bitmaps.
- Since a DIB carries its own color information, color palette management is easier.
- DIBs also make it easy to control gray shades when printing. Any computer running Windows can process DIBs, which are usually stored in BMP disk files or as a resource in your program's EXE or DLL file.

UNIT – II
VISUAL C++ PROGRAMMING – INTRODUCTION

PART – A (2 MARKS)

1. Define Application Framework
2. Define Appwizard
3. Define Classwizard
4. What are the diagnostic tools available in VC++?
5. What are the types of mapping modes?
6. Distinguish between model and modeless dialog controls
7. Define bitmap
8. Mention some of the window common control.
9. What are dialog controls?
10. Mention some of the GDI derived classes.

PART – B

1. Draw & Explain in detail about various components of VC++ (16)
2. Explain in briefly about
a. MM_TEXT Mapping Mode (5)
b. Fixed Scale Mapping Mode (4)
c. Variable Scale Mapping Mode (7)
3. a. Explain in detail about various types of video cards. (10)
b. How to compute Character height (6)
4. What is meant by Modal & Modeless dialog control? Explain Modal dialog controls with a sample programs. (16)
5. a. Discuss about Window Common Controls (12)
b. What are different Mapping Modes available in VC++? (4)
6. a. Explain how to create an instance of color dialog & the functions associated with it. (8)
b. Write a VC++ program to paint the background with a brush. Set the color using the coordinates at which the mouse is clicked. (8)
7. a. Differentiate the modal & modeless dialog (4)
b. Write a VC++ program to create & display a modeless dialog (6)
c. Write a VC++ program to draw a rectangle as the mouse moves (6)

UNIT III

THE DOCUMENT AND VIEW ARCHITECTURE

Document –View Architecture

Global Application object

↓

Execution begins- WinMain by MFC

↓

AfxGetApp() – Gets ptr to Applocation object

↓

InitInstance – Executed(Create Window &assigns

↓

hInstance,nCmdshow –appln object)

↓

RUN()- Implements Message Loop

↓

Terminate if WM_QUIT Message occurs

↓

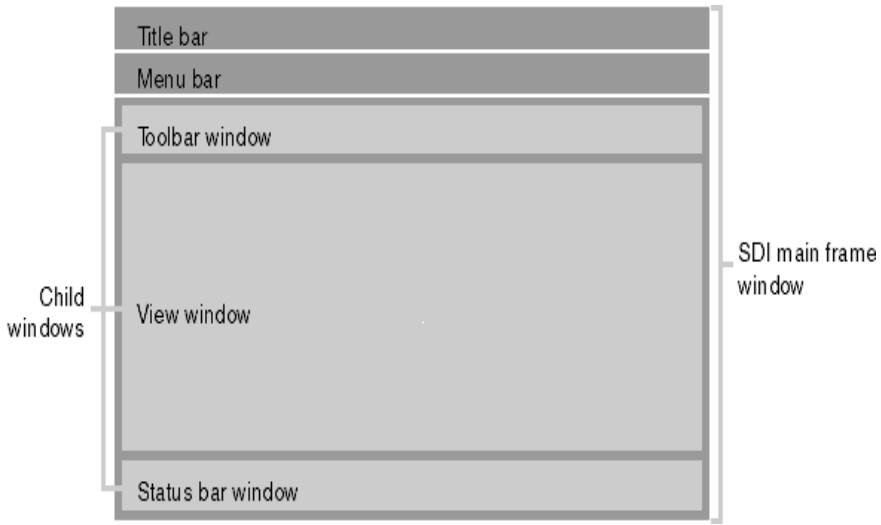
ExitInstance()

Document –View Architecture

Need: Given set of data can have multiple representation- best to separate display from data.
Three Objects:

- Document - Store the data (responsible for reading and writing data)
- View - To display data
- Window - The canvas on which all display take place Menu , Keyboard Accelerators

The Main Frame Window and Document Classes



Application framework- controls interaction between frame and view
MainFrm.h and MainFrm.cpp -application's main frame window class

ex13aDoc.h and ex13aDoc.cpp-application's document class
ex13aView.h and ex13aView.cpp-application's view class

Windows Menus

- 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.
- Menu items - grayed ,have check marks,separator bar.
- Multiple levels pop-up menus are possible.
- Each Menu item – ID which is defined in resource.h
- Entire resource definition - .rc file(resource script file)
- Command Processing
- WM_COMMAND message – menu selection, keyboard accelerators,toolbar and dialog button clicks by windows.
- Message Map Entry:

ON_COMMAND(ID, command handler)
Id of menu clicked item corresponding message handler
For example, Menu item is Line (ID is
ID_LINE)
For handle it in the view class,
In view.cpp

BEGIN_MESSAGE_MAP(CMyView, CView) ON_COMMAND(ID_LINE, OnLine)
END_MESSAGE_MAP()
void CMyView::OnLine()
{ // command message processing code
}

In MyView.h – Give the definition
afx_msg void OnZoom();
before DECLARE_MESSAGE_MAP() macro

Command update handler function.

- Whenever a pop-up menu is first displayed or menu item is clicked ,MFC calls this function.
- The handler function's argument is a *CCmdUI* object, which contains a pointer to the corresponding menu item.
- this pointer to modify the menu item's appearance by its operations such as enable , setcheck etc.

In view.cpp

BEGIN_MESSAGE_MAP(CMyView, CView) ON_UPDATE_COMMAND_UI(ID_LINE, OnUpdateLine) END_MESSAGE_MAP()

void CMyView::OnUpdateLine()
{ // command message processing code
}

In MyView.h – Give the definition
afx_msg void OnUpdateLine();

MFC Text Editing features

Edit Control and Rich Edit Control: CEditView and CRichEditView

CEditView Class:

- Maximum size is 64 KB, work in View and Edit classes
- can't mix font and cut copy paste is possible.

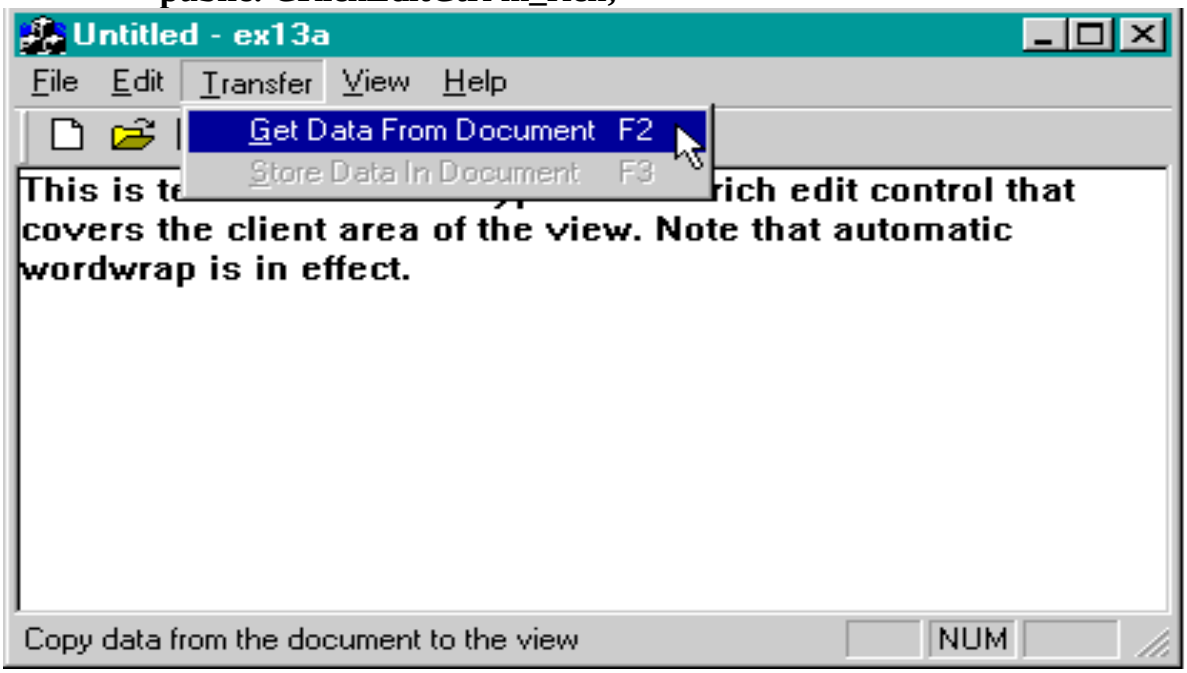
CRichEditView

- Supports mixed format and large quantities of text.
- can include embedded OLE objects
- **CRichEditView** maintains the text and formatting characteristic of text.
- **CRichEditDoc** maintains the list of OLE client items which are in the view.
- **CRichEditCntrItem** provides container-side access to the OLE client item
- Example

1. **Add a *CString* data member to the *CEx13aDoc* class. In ex13aDoc.h**

   public: CString m_strText;
2. **Add a *CRichEditCtrl* data member to the *CEx13aView* class. In file ex13aView.h**

   public: CRichEditCtrl m_rich;



In ex13aDoc.cpp

```
1. void CEx13aDoc::OnEditClearDocument()
                        { m_strText.Empty(); }
2. void CEx13aDoc::OnUpdateEditClearDocument
                                (CCmdUI* pCmdUI)
                                    {
                                     pCmdUI->Enable(!m_strText.IsEmpty());
                                    }
```

In CEx13View.cpp : Creation RichEdit Control

- int CEx13aView::OnCreate(LPCREATESTRUCT lpCreateStruct)

```
{     CRect rect(0, 0, 0, 0);
 if (CView::OnCreate(lpCreateStruct) == -1) return -1;
```

```
m_rich.Create(ES_AUTOVSCROLL | ES_MULTILINEES_WANTRETURN |
WS_CHILD | WS_VISIBLE | WS_VSCROLL, rect, this, 1);
 return 0;
 }
```

2. void CEx13aView::OnSize(UINT nType, int cx, int cy)
{
CRect rect;
CView::OnSize(nType, cx, cy); GetClientRect(rect); m_rich.SetWindowPos(&wndTop, 0, 0, rect.right - rect.left, rect.bottom - rect.top, SWP_SHOWWINDOW);
}
position &structure

3. void CEx13aView::OnTransferGetData()
{
CEx13aDoc* pDoc = GetDocument();
m_rich.SetWindowText(pDoc- >m_strText);
m_rich.SetModify(FALSE);
}

4. void CEx13aView::OnTransferStoreData()
{
CEx13aDoc* pDoc = GetDocument(); m_rich.GetWindowText(pDoc->m_strText); m_rich.SetModify(FALSE);
}

5. void CEx13aView::OnUpdateTransferStoreData
(CCmdUI* pCmdUI)
{
pCmdUI->Enable(m_rich.GetModify());
}

6. Build and Run the application

Creating Floating Pop-Up Menus

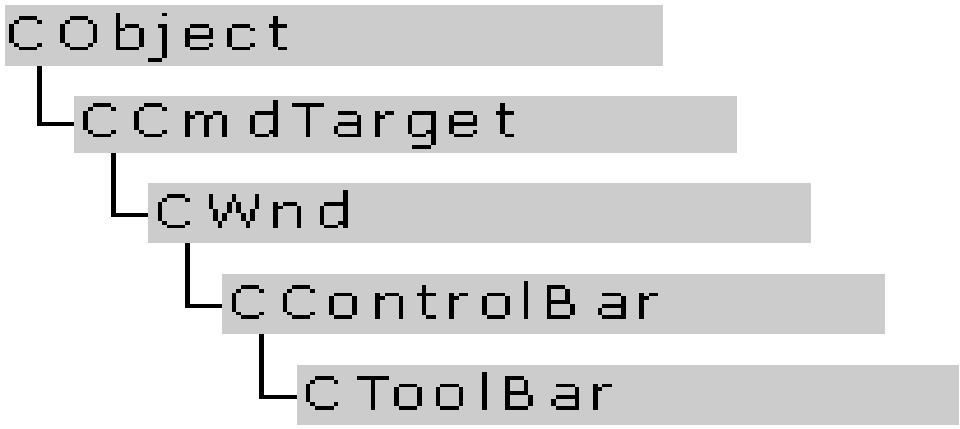
```
void CMyView::OnContextMenu(CWnd *pWnd,
                            CPoint point)
        {
CMenu menu; menu.LoadMenu(IDR_MYFLOATINGMENU); menu.GetSubMenu(0) ->TrackPopupMenu
 (TPM_LEFTALIGN | TPM_RIGHTBUTTON,
  point.x, point.y, this);
}
```

- **Extended Command Processing**
- BEGIN_MESSAGE_MAP(CMyView, CView)
ON_COMMAND_EX_RANGE(IDM_ZOOM_1,

IDM_ZOOM_2, OnZoom)
END_MESSAGE_MAP()
- ***ON_COMMAND_RANGE(id1, id2, Fxn)***
- *ON_COMMAND_EX_RANGE*
- *ON_UPDATE_COMMAND_UI_RANGE*

ToolBar & StatusBar

ToolBar



- A toolbar consists of a number of horizontally (or vertically) arranged graphical buttons that might be clustered in groups
- Pressing a toolbar button is equivalent to choosing a menu item(WM_COMMAND messages).
- An update command UI message handler is used to update the button's state
- MFC toolbar can “dock” to any side of its parent window or float in its own mini-frame window.
- you can change its size and drag it.
- A toolbar can also display tool tips as the user moves the mouse over the toolbar’s buttons.

ToolBar Bitmap:

-Each button on a toolbar appears to have its own bitmap,
but actually a single bitmap serves the entire toolbar.
-has tile, 15 pixels high and 16 pixels wide
-The toolbar bitmap is stored in the file Toolbar.bmp

- **Button State:**

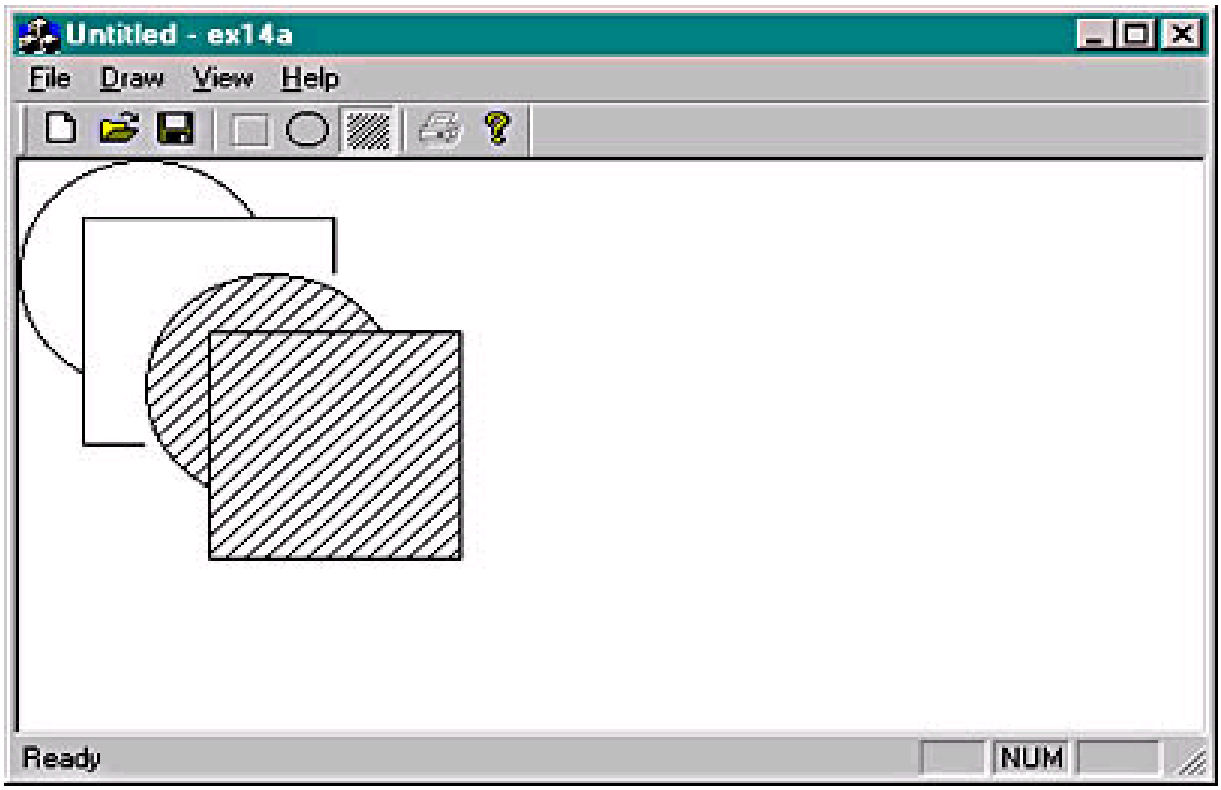
0: Normal, unpressed state.
TBSTATE_CHECKED Checked (down) state.*TBSTATE_ENABLED* Available for use. Button is grayed and unavailable if this state is not set.
TBSTATE_HIDDEN Not visible.
TBSTATE_INDETERMINATE Grayed.
TBSTATE_PRESSED Currently selected (pressed) with the mouse.
TBSTATE_WRAP Line break follows the button

Locating the Main FrameWindow

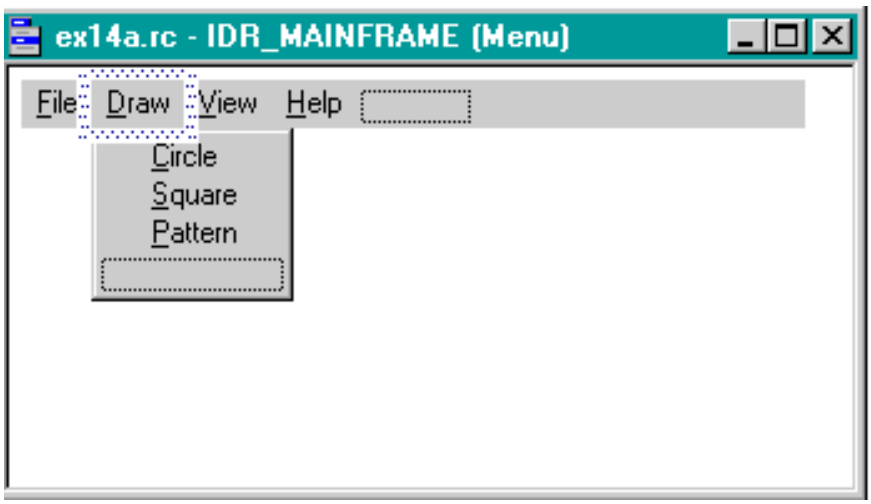
- The toolbar and status bar objects you'll be working with are attached to the application's main frame window, not to the view window
- find the main frame window through the application object.

```
CMainFrame* pFrame = (CMainFrame*)
                     AfxGetApp()->m_pMainWnd;
CToolBar* pToolBar = &pFrame->m_wndToolBar;
```

Example



- RunAppWizard to create an SDI application
  & **Use the resource editor to edit the application's main menu as follows**



3. **Use the resource editor to update the application's toolbar-Edit the I*DR_MAINFRAME* toolbar resource**
4. **Give the ID to each Button**
5. **USE ClassWizard to add command and update command UI messages for ID_CIRCLE,ID_SQUARE &ID_PATTERN**

6. **In the file ex14aView.h,**
   private:
   CRect m_rect;
   BOOL m_bCircle;
   BOOL m_bPattern;

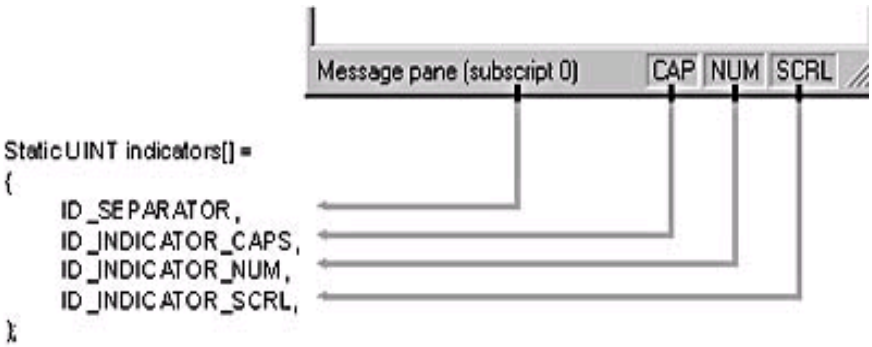
7. void CEx14aView::OnDrawCircle()
   {
   m_bCircle = TRUE;

```
        m_rect += CPoint(25, 25);
         InvalidateRect(m_rect);      }
8. void CEx14aView::OnDrawSquare()
          { m_bCircle = FALSE;
             m_rect += CPoint(25, 25);
            InvalidateRect(m_rect);
            }
9. void CEx14aView::OnDrawPattern()    //toggles
            { m_bPattern ^= 1; }
10. void CEx14aView::OnUpdateDrawCircle (CCmdUI*   pCmdUI)
             { pCmdUI->Enable(!m_bCircle); }
11.voidCEx14aView::OnUpdateDrawSquare(CCmdUI*  pCmdUI)
        { pCmdUI->Enable(m_bCircle); }
12. void CEx14aView::OnUpdateDrawPattern(CCmdUI* pCmdUI)
            { pCmdUI->SetCheck(m_bPattern); }
```

StatusBar

- neither accepts user input nor generates command messages
- to display text in panes under program control
- supports two types of text panes
    - message line panes
    - status indicator panes

- **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

Get access to the status bar object

```
CMainFrame* pFrame = (CMainFrame*) AfxGetApp()->
                          m_pMainWnd;
CStatusBar* pStatus = &pFrame->m_wndStatusBar;
```

Display String in Massage Line-SetPaneText

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

Pane No.:

0-leftmost pane

1-next pane to the right and so forth.

The Status Indicator

- 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.
- –Contains Indicators .
- An indicator is identified by a string resource ID.
- same ID is used to route update command UI messages.
- For example Caps Lock indication by status bar

In Mainframe.cpp

- ON_UPDATE_COMMAND_UI(ID_INDICATOR_CAPS,

  OnUpdateKeyCapsLock)
- void MainFrame::OnUpdateKeyCapsLock(CCmdUI* pCmdUI)

  { pCmdUI->Enable(::GetKeyState(VK_CAPITAL) & 1); }

For Left Button Status

- ON_UPDATE_COMMAND_UI(ID_LEFT, OnLeft)
- void MainFrame::OnLeft(CCmdUI* pCmdUI)

  { pCmdUI->Enable(::GetKeyState(VK_LBUTTON) & 1); }
- **InMainframe.h**

  void Onleft(CCmdUI* j);
- **Taking Control of the Status Bar**

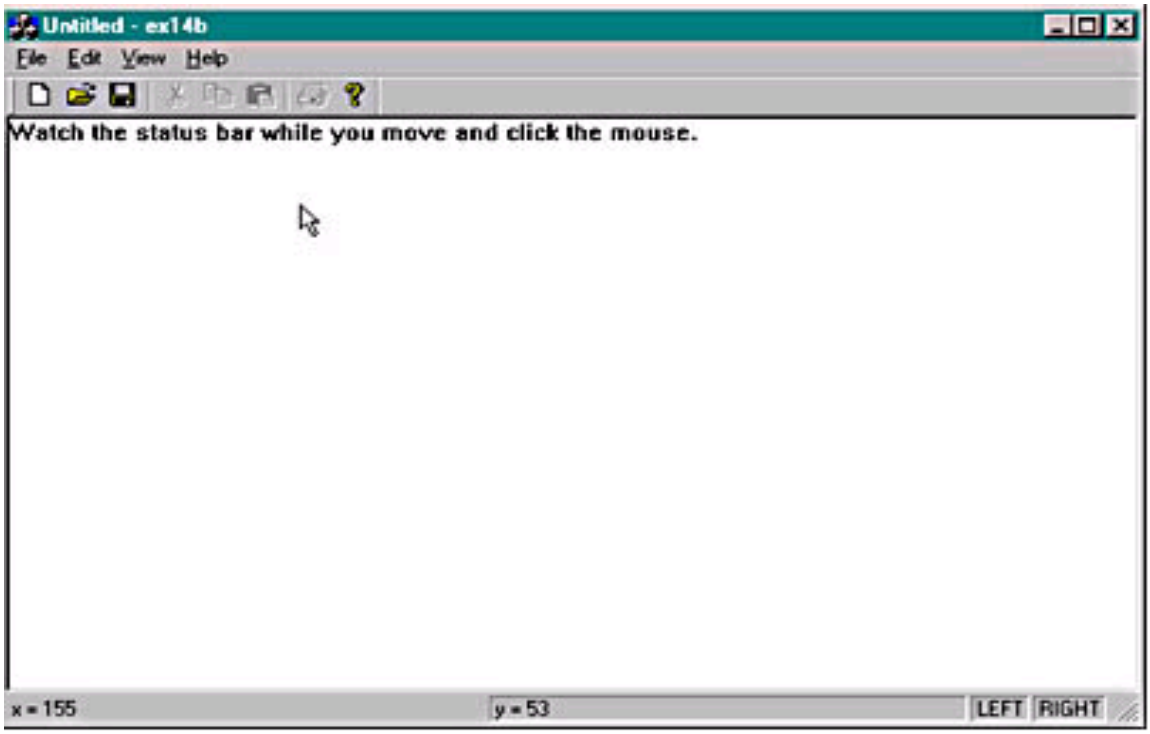
Avoid Default status bar and have your own status bar

- To assign your own ID, you must replace this call

m_wndStatusBar.Create(this); with this call

- m_wndStatusBar.Create(this, WS_CHILD | WS_VISIBLE | CBRS_BOTTOM, ID_MY_STATUS_BAR);

Example



1. Use the string editor to edit the application's string table resource.
ID Caption
ID_INDICATOR_LEFT **-LEFT**
ID_INDICATOR_RIGHT **-RIGHT**
2. Choose Resource Symbols from the View menu. Add the new status bar identifier,
ID_MY_STATUS_BAR,
and accept the default value.
3. In MainFrame.cpp
void CMainFrame::OnUpdateLeft(CCmdUI* pCmdUI)
{ pCmdUI->Enable(::GetKeyState(VK_LBUTTON) < 0); }
4. void CMainFrame::OnUpdateRight(CCmdUI* pCmdUI)
{ pmdUI->Enable(::GetKeyState(VK_RBUTTON) < 0); }
5. In MainFrame.cpp MessageMap Entry
ON_UPDATE_COMMAND_UI(ID_INDICATOR_LEFT,
OnUpdateLeft) ON_UPDATE_COMMAND_UI(ID_INDICATOR_RIGHT,
OnUpdateRight)
6. In MainFrm.h.
afx_msg void OnUpdateLeft(CCmdUI* pCmdUI);
afx_msg void OnUpdateRight(CCmdUI*pCmdUI);
7. Edit the MainFrm.cpp file.
Replace the original *indicators* array with the following boldface code:
static UINT indicators[] =
{ID_SEPARATOR, // first message line pane ID_SEPARATOR, // second message
line pane ID_INDICATOR_LEFT,
ID_INDICATOR_RIGHT };
8. **Use ClassWizard to add View menu command handlers in the class *CMainFrame.***
1. ID_VIEW_STATUS_BAR **- COMMAND**
-OnViewStatusBar
2. ID_VIEW_STATUS_BAR--
UPDATE_COMMAND_UI
-OnUpdateViewStatusBar
void CMainFrame::OnViewStatusBar()

{ **m_wndStatusBar.ShowWindow((m_wndStatusBar.GetStyle() & WS_VISIBLE) == 0);**
RecalcLayout();
}
void CMainFrame::OnUpdateViewStatusBar(CCmdUI* pCmdUI)
{ **pCmdUI- >SetCheck((m_wndStatusBar.GetStyle() & WS_VISIBLE) != 0);** }

***9.In OnCreate* member function**

Replace
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 }
with the statement shown here:

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

10. InView.cpp

void CEx14bView::OnDraw(CDC* pDC)
{ **pDC->TextOut(0, 0, "Watch the status bar while you move and click the mouse.");** }

11. void CEx14bView::OnMouseMove(UINT nFlags, Cpoint
point)
{ **CString str; CMainFrame* pFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;**
CStatusBar* pStatus = &pFrame->m_wndStatusBar;
if (pStatus)
{ str.Format("x = %d", point.x);
pStatus->SetPaneText(0, str);
str.Format("y = %d", point.y);
pStatus->SetPaneText(1, str); }}

12 #include "MainFrm.h"

- **A Reusable Frame Window Base Class**
- *CString* class.
- Build your own reusable base class .
- Access to the Windows Registry.
- PreCreateWindow & ActiveFrame function.

CString class

- dynamic memory allocation-*const char**.
- CString strFirstName("Elvis");
- CString strLastName("Presley");
- CString strTruth = strFirstName + " " +
strLastName;
- strTruth += " is alive";

int nError = 23;
CString strMessageText;
strMessageText.Format("Error number %d", nError);

AfxMessageBox(strMessageText);

- CString strTest("test");
- strncpy(strTest, "T", 1);

CString::GetBuffer - "locks down" the buffer with a specified size and returns a *char**.
ReleaseBuffer - to make the string dynamic again.
CString strTest("test"); strncpy(strTest.GetBuffer(5), "T", 1); strTest.ReleaseBuffer();

- Build your own reusable base class
- *CPersistentFrame* - derived from the *CFrameWnd*

- 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 Windows Registry**
- is a set of system files, managed by Windows, in which Windows and individual applications can store and access permanent information.
- is organized as a kind of hierarchical database in which string and integer data is accessed by a multipart key.
- TEXTPROC

Text formatting
Font = Times Roman
Points = 10

- The *SetRegistryKey* function's string parameter establishes the top of the hierarchy,

SetRegistryKey(" TEXTPROC ");
Following Registry functions(CWinApp) define the bottom two levels: called heading name and entry name.

- *GetProfileInt*
- *WriteProfileInt*
- *GetProfileString*
- *WriteProfileString*

```
AfxGetApp()->WriteProfileString("Text formatting", "Font",
                                "Times Roman");
AfxGetApp()->WriteProfileInt("Text formatting", "Points", 10);
```

- ***ActivateFrame* Member Function** *CFrameWnd*
- The key to controlling the frame's size

- 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).
- The framework's job is to call the *CWnd::ShowWindow* function with the parameter *nCmdShow*. The *nCmdShow* parameter determines whether the window is maximized or minimized or both.
- Override *ActivateFrame* in your derived frame class, to change the value of *nCmdShow* before passing to it .
- *CWnd::SetWindowPlacement* function, which sets the size and position of the frame window, and you can set the visible status of the control bars.
- First time call ie *CPersistentFrame::ActivateFrame* function operates only when the application starts.
- **The *PreCreateWindow* Member Function-CWnd**
- to change the characteristics of your window before it is displayed
- calls this function before it calls *ActivateFrame*.
- a *CREATESTRUCT* structure as a parameter, and two of the data members in this structure are *style* and *dwExStyle*.
- *CREATESTRUCT* member *lpszClass* is also useful to change the window's background brush, cursor, or icon.

```
BOOL MyView::PreCreateWindow(CREATESTRUCT& c)
{     …..
      ….
 c.lpszClass =AfxRegisterWndClass(CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW, AfxGetApp()-
> LoadCursor(IDC_MYCURSOR), ::CreateSolidBrush (RGB(255, 0, 0)));
if (cs.lpszClass != NULL) { return TRUE; }
  else { return FALSE; }
}
```

Style flag -determines whether the window has a border,
scroll bars, a minimize box, and so on.
Extended style – double border, left aligned border etc

- Document View Architecture
- Menu, Keyboard Accelerators
- Status Bar, ToolBar
- Reusable Frame Window Class
- Global Application object
- Execution begins- WinMain by MFC
- AfxGetApp() – Gets ptr to Applocation object
- InitInstance – Executed(Create Window &assigns
- hInstance,nCmdshow –appln object)
- RUN()- Implements Message Loop

- Terminate if WM_QUIT Message occurs
- ExitInstance()
- Separating Document from its View
- Need: Given set of data can have multiple

  representation- best to separate display
  from data.
- Three Objects:
    - Document - Store the data (responsible for

  reading and writing data)
    - View - To display data(each view –one document)
    - Window - The canvas on which all display

  take place
- **Document-View Interaction Functions**
- **The *CView::GetDocument* Function-provides the**

  document pointer that can be used to access document class member functions or public data embers.
  CEx13aDoc* pDoc = GetDocument();
  m_rich.SetWindowText(pDoc- >m_strText);
- **The *CDocument::GetNextView* -navigates from the**

  document to the view, but because a document can have more than one view, it's necessary to call this member function once for each view, inside a loop
- ***CDocument::UpdateAllViews***

-If the document data changes for any reason, all views must be notified so that they can update their representations of that data.

-GetDocument()->UpdateAllViews(NULL): Update all associated view .

-GetDocument()->UpdateAllViews(this): - Update all associated view except current.

Syntax: void UpdateAllViews(CView* *pSender*, LPARAM *lHint* = 0L, CObject* *pHint* = NULL);

Parameters

***pSender*-Points to the view that modified the document, or NULL if all views are to be updated.**

***lHint* - Contains information about the modification.**

***pHint* -Points to an object storing information about the modification.**

CView::OnUpdate:

- virtual function is called by the application framework in response to your application's call to the C*Document::UpdateAllViews* function.
- view class's *OnUpdate* function accesses the document, gets the document's data, and then updates the view's data members or controls to reflect the changes.
- *OnUpdate* can invalidate a portion of the view, causing the view's *OnDraw* function to use document data to draw in the window.

- Same parameters as UpdateAllViews.

CView::OnInitialUpdate

-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.

-calls *OnUpdate.*

-use your derived class's OnInitialUpdate function to initialize your view object

-When the application starts, the application framework calls *OnInitialUpdate* immediately after *OnCreate*

CDocument::OnNewDocument

-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.

-to set the initial values of your document's data members.

Application starts

- ***CMyDocument* object constructed**
- ***CMyView* object constructed**
- **View window created**
- ***CMyView::OnCreate* called (if mapped)**
- ***CMyDocument::OnNewDocument* called**
- ***CMyView::OnInitialUpdate* called**
- **View object initialized**
- **View window invalidated**
- ***CMyView::OnDraw* called**

User edits data

***CMyView* functions update *CMyDocument* data members**

User exits application

***CMyView* object destroyed**

Frame Window - Definition

- An application or some parts of an application when framed by Windows are called Frame Windows
- Frame windows act as containers for other windows such as control bars or child controls.

Basic types of Frame Windows

- Single Document Interface (SDI) frame windows
- Multiple Document Interface (MDI) frame windows.
- MDI frame windows can contain MDI child windows

Serialization

- Serialization is an important concept in MFC programming because it is the basis for MFC's ability to open and save documents in document/view applications.

Serialization Process

- when someone using a document/view application selects Open or Save from the application's File menu, MFC opens the file for reading or writing and passes the application a reference to a *CArchive* object.
- The application, in turn, serializes its persistent data to or from the archive and, by so doing, saves a complete document to disk or reads it back again. A document whose persistent data consists

entirely of primitive data types or serializable objects can often be serialized with just a few lines of code.

Serialization - Write

- Assume that a *CFile* object named *file* represents an open file, that the file was opened with write access, and that you want to write a pair of integers named *a* and *b* to that file. One way to accomplish this is to call *CFile::Write* once for each integer:
- *file.Write (&a, sizeof (a));*
- *file.Write (&b, sizeof (b));*
- An alternative method is to create a *CArchive* object, associate it with the *CFile* object, and use the << operator to serialize the integers into the archive:
- *CArchive ar (&file, CArchive::store);*
  ar << a << b;

Serialization – Read (deserialize)

- Assuming *file* once again represents an open file and that the file is open with read access, the following code snippet attaches a *CArchive* object to the file and reads, or *deserializes*, the integers from the file:
- *CArchive ar (&file, CArchive::load);*
  - *ar >> a >> b;*
- MFC allows a wide variety of primitive data types to be serialized this way, including BYTEs, WORDs, LONGs, DWORDs, floats, doubles, ints, unsigned ints, shorts, and chars.
- SDI and MDI
- MFC supports two types of document/view applications. *Single document interface* (SDI) applications support just one open document at a time. *Multiple document interface* (MDI) applications permit two or more documents to be open concurrently and also support multiple views of a given document. The WordPad applet is an SDI application; Microsoft Word is an MDI application.

SDI

- SDI application frame windows are derived from the class CFrameWnd.
- The MFC library supports two distinct application types: Single Document Interface (SDI) and Multiple Document Interface (MDI). An SDI application has, only one window. If the application depends on disk-file "documents," only one document can be loaded at a time. The original Windows Notepad is an example of an SDI application.

The standard SDI frame menus

- *The child windows within an SDI main frame window*
- SDI Document View Architecture
- SDI Application

Startup steps in a Microsoft Windows MFC library application:

- Windows loads your program into memory.
- The global object *theApp* is constructed. (All globally declared objects are constructed immediately when the program is loaded.)
- Windows calls the global function *WinMain*, which is part of the MFC library. (*WinMain* is equivalent to the non-Windows *main* function—each is a main program entry point.)
- Steps…
- *WinMain* searches for the one and only instance of a class derived from *CWinApp*.
- *WinMain* calls the *InitInstance* member function for *theApp*, which is overridden in your derived application class.
- Your overridden *InitInstance* function starts the process of loading a document and displaying the main frame and view windows.
- *WinMain* calls the *Run* member function for *theApp*, which starts the processes of dispatching window messages and command messages.

Object Relationship

- Steps for processing SDI *InitInstance*
- Create an SDI document template from MFC's *CSingleDocTemplate* class.
- Adding to list of document templates
- Initializing command line info values

1. Processing command line parameters
2. Displaying applications frame window

InitInstance function for an SDI application generated by AppWizard

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate
 (IDR_MAINFRAME,
  RUNTIME_CLASS (CMyDoc),
 RUNTIME_CLASS (CMainFrame),
 RUNTIME_CLASS (CMyView)
 );

 AddDocTemplate (pDocTemplate);
  ……..
  ……..
 CCommandLineInfo cmdInfo;
 ParseCommandLine (cmdInfo);
 if (!ProcessShellCommand (cmdInfo)) return FALSE;
m_pMainWnd->ShowWindow (SW_SHOW);
m_pMainWnd->UpdateWindow ();
```

1. create an SDI document template from *CSingleDocTemplate class*

```
  CSingleDocTemplate* pDocTemplate;
   pDocTemplate = new CSingleDocTemplate
  ( IDR_MAINFRAME,
    RUNTIME_CLASS (CMyDoc),
    RUNTIME_CLASS (CMainFrame),
    RUNTIME_CLASS (CMyView)
  );
```

1. The template's constructor was passed four parameters: an integer value equal to IDR_MAINFRAME and three RUNTIME_CLASS pointers.
2. AppWizard uses the resource ID IDR_MAINFRAME in the code that it generates. The RUNTIME_CLASS macro surrounding the class names returns a pointer to a CRuntimeClass structure for the specified class, which enables the framework to create objects of that class at run time.
3. Adding to list of document templates
4. After the document template is created, the statement
5. *AddDocTemplate (pDocTemplate);*
6. adds it to the list of document templates maintained by the application object. Each template registered in this way defines one document type the application supports. SDI applications register just one document type
7. Initialize command line info values
8. The statements
9. *CCommandLineInfo cmdInfo; Par*
10. *seCommandLine (cmdInfo);*

11. use *CWinApp::ParseCommandLine* to initialize a *CCommandLineInfo* object with values reflecting the parameters entered on the command line, which often include a document file name.
12. Process command line parameters
13. The statements
14. *if (!ProcessShellCommand (cmdInfo)) return FALSE;*
15. "process" the command line parameters. Among other things, P*rocessShellCommand* calls *CWinApp::OnFileNew* to start the application with an empty document if no file name was entered on the command line, or *CWinApp::OpenDocumentFile* to load a document if a document name was specified. It's during this phase of the program's execution that the framework creates the document, frame window, and view objects using the information stored in the document template. *ProcessShellCommand* returns TRUE if the initialization succeeds and FALSE if it doesn't.
16. Display applications frame window
17. If initialization is successful, the statements
18. *m_pMainWnd->ShowWindow (SW_SHOW); m_pMainWnd->UpdateWindow ();*
19. display the application's frame window (and by extension, the view) on the screen.
20. *Routing of command messages sent to an SDI frame window*.

Example Program - The SdiSquares Application

The program shown in Figure is an SDI document/view application that displays a grid of squares four rows deep and four columns wide. Initially, each square is colored white. However, you can change a square's color by clicking it with the left mouse button. By default, clicking changes a square's color to red. You can select alternate colors from the Color menu and thereby create a multicolored grid containing squares of up to six different colors.

The SdiSquares Application…

Use AppWizard to create a new project named *SdiSquares*. In AppWizard's Step 1 dialog box, choose Single Document as the application type and check the Document/View Architecture Support box, as shown in Figure

The SdiSquares Application…

In the Step 3 dialog box, uncheck the ActiveX Controls box.

In Step 4, uncheck Docking Toolbar, Initial Status Bar, Printing And Print Preview, and 3D Controls.

The SdiSquares Application…

Also in the Step 4 dialog box, click the *Advanced button* and type the letters *sqr* into the File Extension box (as shown in Figure) to define the default file name extension for SdiSquares documents.

The SdiSquares Application…

1. In the Step 6 dialog box, manually edit the class names as *CSquaresApp*. Everywhere else, accept the AppWizard defaults.

The SdiSquares Application…

Add the member variables *m_clrGrid* and *m_clrCurrentColor* to the document class, and add code to initialize them to *OnNewDocument*. AppWizard overrides *OnNewDocument*, so all you have to do is add the statements that initialize the data members.

The SdiSquares Application…

Add the member functions *GetCurrentColor*, *GetSquare*, and *SetSquare* to the document class. Be sure to make them public member functions, since they must be accessible to the view.

Modify the S*erialize* function that AppWizard included in the document class to serialize *m_clrGrid* and *m_clrCurrentColor*.

Implement the view's *OnDraw* function. AppWizard generates a do-nothing *OnDraw* function; you write the code to perform application-specific duties.

The SdiSquares Application…
Add the WM_LBUTTONDOWN handler (*OnLButtonDown*) to the view. You can add the message handler by hand or use ClassWizard to add it. I used ClassWizard.
Open the AppWizard-generated application menu for editing, delete the Edit menu, and add the Color menu. Then write command and update handlers for the new menu items. As with message handlers, you can add command and update handlers manually or you can add them with ClassWizard's help.

SDI vs MDI

MDI

- MDI application frame windows are derived from the class CMDIFrameWnd.
- An MDI application has multiple child windows, each of which corresponds to an individual document. Microsoft Word is a good example of an MDI application. When you run AppWizard to create a new project, MDI is the default application type

The parent-child hierarchy of a Windows MDI application.

- MDI application
- MDI windows
- MDI classes

An MDI application has two frame window classes and many frame objects

Base Class

CMDIFrameWnd
CMDIChildWnd

AppWizard-Generated Class

CMainFrame
CChildFrame
The MDI frame-view window relationship

- Splitter Windows
- A splitter window is a window that can be divided into two or more panes horizontally, vertically, or both horizontally and vertically using movable splitter bars. Each pane contains one view of a document's data. The views are children of the splitter window, and the splitter window itself is normally a child of a frame window.
- Splitter windows
- Using splitter windows provided by MFC, a single document interface (SDI) application can present two or more views of the same document in resizeable "panes" that subdivide the frame window's client area
- In an SDI application, the splitter window is a child of the top-level frame window.
- In an MDI application, the splitter window is a child of an MDI document frame.
- Types of splitter windows
- static splitter window:

The numbers of rows and columns in a static splitter window are set when the splitter is created and can't be changed by the user. The user is, however, free to resize individual rows and columns. A static

splitter window can contain a maximum of 16 rows and 16 columns. For an example of an application that uses a static splitter, look no further than the Windows Explorer. Explorer's main window is divided in half vertically by a static splitter window.

- Static splitter window
- Dynamic Splitter window
- A dynamic splitter window is limited to at most two rows and two columns, but it can be split and unsplit interactively. The views displayed in a dynamic splitter window's panes aren't entirely independent of each other: when a dynamic splitter window is split horizontally, the two rows have independent vertical scroll bars but share a horizontal scroll bar. Similarly, the two columns of a dynamic splitter window split vertically contain horizontal scroll bars of their own but share a vertical scroll bar. The maximum number of rows and columns a dynamic splitter window can be divided into are specified when the splitter is created. Thus, it's a simple matt
- Dynamic Splitter window
- Procedure for Creating and initializing a dynamic splitter window

1. Add a *CSplitterWnd* data member to the frame window class.
2. Override the frame window's virtual *OnCreateClient* function, and call *CSplitterWnd::Create* to create a dynamic splitter window in the frame window's client area.

- Creating splitter window…
- Assuming *m_wndSplitter* is a *CSplitterWnd* object that's a member of the frame window class *CMainFrame*, the following *OnCreateClient* override creates a dynamic splitter window inside the frame window:

```
BOOL CMainFrame::OnCreateClient
(LPCREATESTRUCT lpcs,
CCreateContext* pContext)
{
 return m_wndSplitter.Create (this, 2, 1, CSize (1, 1), pContext);
}
```

- Creating splitter window…
- The first parameter to *CSplitterWnd::Create* identifies the splitter window's parent, which is the frame window.
- The second and third parameters specify the maximum number of rows and columns that the window can be split into.
- Because a dynamic splitter window supports a maximum of two rows and two columns, these parameter values will always be 1 or 2.
- The fourth parameter specifies each pane's minimum width and height in pixels. The framework uses these values to determine when panes should be created and destroyed as splitter bars are moved.
- Creating splitter window…
- *CSize* values equal to (1,1) specify that panes can be as little as 1 pixel wide and 1 pixel tall.

- The fifth parameter is a pointer to a *CCreateContext* structure provided by the framework. The structure's *m_pNewViewClass* member identifies the view class used to create views in the splitter's panes.
- The framework creates the initial view for you and puts it into the first pane. Other views of the same class are created automatically as additional panes are created.

UNIT – III
THE DOCUMENT VIEW ARCHITECTURE

PART – A (2 MARKS)

1. Define Keyboard Accelerator
2. List out Rich Edit Control Functions
3. Define toolbar
4. List out toolbar states.
5. Define Statusbar
6. Define Status Indicator
7. What are the two text editing tools?
8. What are the steps to be followed to build floating popup menus?
9. What are the characteristic of SDI frame window?
10. Define Serialization
11. Explain splitter window?
12. Distinguish between dynamic and static splitter windows
13. Define Document – View Architecture
14. Distinguish Implicit and Explicit Linkage
15. What is LoadLibrary function?

PART – B

1. Write down the steps to create a VC++ program that encapsulates the menu, keyboard accelerator and tool bar to draw a circle and rectangle and show the output. (16)
2. What are the functions performed in SDI application and Explain that functions in detail (16)
3. Write down the steps to create a VC++ program to create an Extension DLL and use it and test it in the client program. (16)
4. Develop a dialog based application to simulate a calculator. The calculator should add, multiply, subtract and divide 2 integers. (16)
5. Develop a DLL to add & multiply two numbers and write an application to use the DLL (16)
6. Explain how to create a toolbox for the application. (16)
7. Explain SDI & MDI application in detail. (16)
8. a. What is Rich Edit control & Discuss the supporting MFC classes for the control. (8)
   b. Discuss the Menu item properties (8)

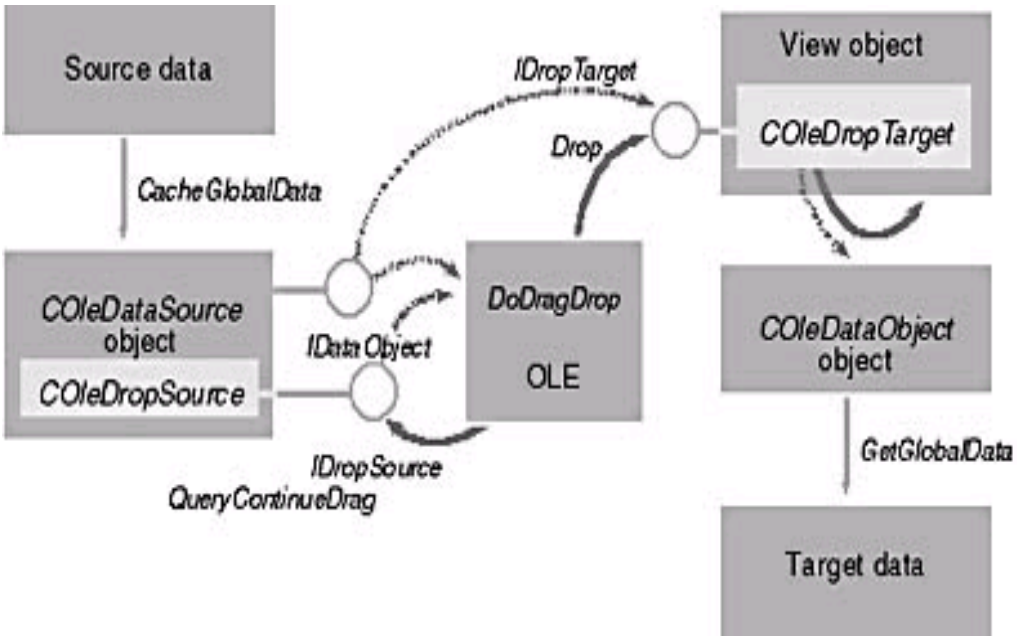
UNIT IV
ACTIVEX AND OBJECT LINKING AND EMBEDDING (OLE)

MFC Drag and Drop

- Drag and drop was the ultimate justification for the data object code you've been looking at.
- OLE supports this feature with its *IDropSource* and *IDropTarget* interfaces plus some library code that manages the drag-and-drop process.
- The MFC library offers good drag-and-drop support at the view level, so we'll use it.
- Drag-and-drop transfers are immediate and independent of the clipboard.
- If the user cancels the operation, there's no "memory" of the object being dragged.
- Drag-and-drop transfers should work consistently between applications, between windows of the same application, and within a window.
- When the user starts the operation, the cursor should change to an arrow_rectangle combination.
- If the user holds down the Ctrl key, the cursor turns into a plus sign (+), which indicates that the object is being copied rather than moved.
- MFC also supports drag-and-drop operations for items in compound documents.
- This is the next level up in MFC OLE support, and it's not covered in this chapter.
- Look up the OCLIENT example in the online documentation under Visual C++ Samples.
- ***The Source Side of the Transfer***
- ***The Destination Side of the Transfer***

OnDragEnter Adjusts the focus rectangle and then calls O*nDragOver* *OnDragOver* Moves the dotted focus rectangle and sets the drop effect (determines cursor shape)
OnDragLeave Cancels the transfer operation; returns the rectangle to its original position and size
OnDrop Adjusts the focus rectangle and then calls the *DoPaste* helper function to get formats from the data object

The Drag-and-Drop Sequence

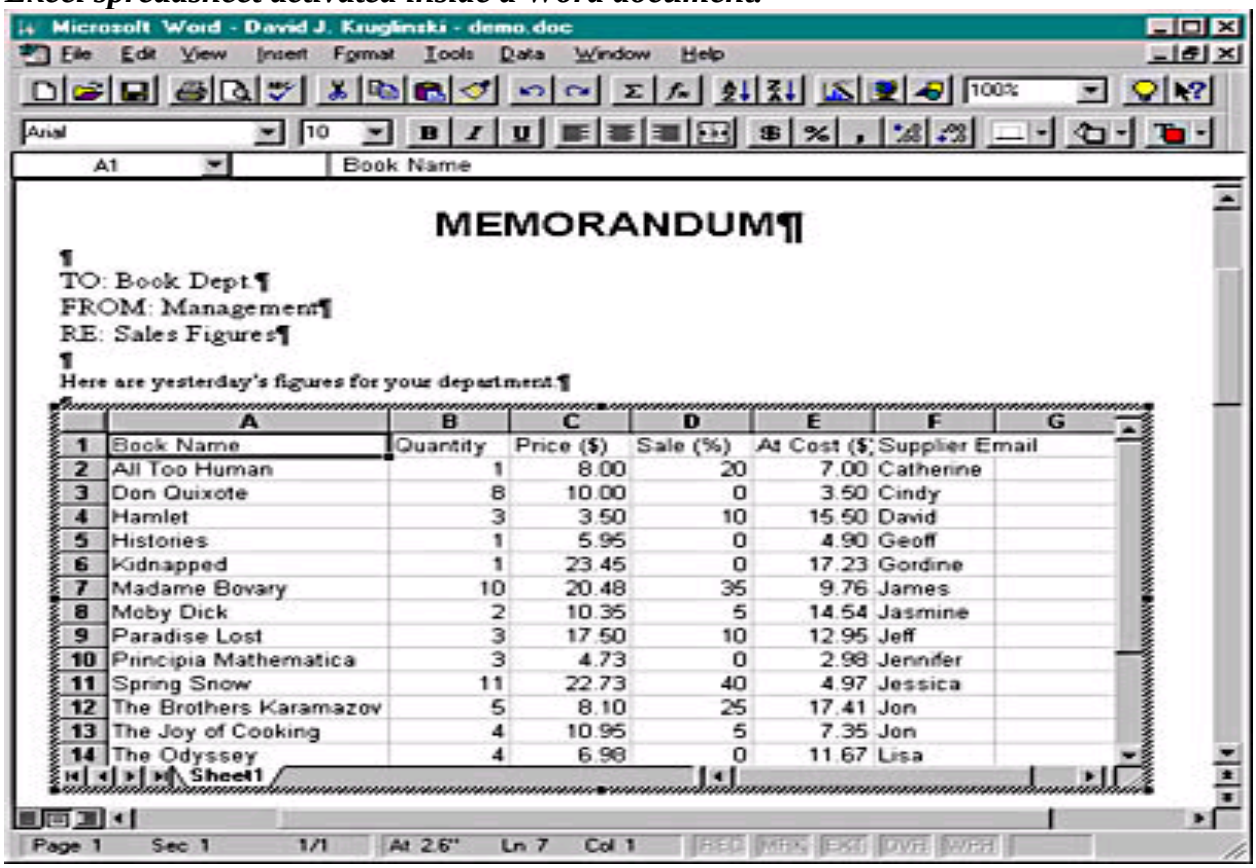


1. User presses the left mouse button in the source view window.
2. Mouse button handler calls *CRectTracker::HitTest* and finds out that the cursor was inside the tracker rectangle.
3. Handler stores formats in a *COleDataSource* object.
4. Handler calls *COleDataSource::DoDragDrop* for the data source.
5. User moves the cursor to the view window of the target application.
6. OLE calls *IDropTarget::OnDragEnter* and *OnDragOver* for the *COleDropTarget* object, which calls the corresponding virtual functions in the target's view. The *OnDragOver* function is passed a *COleDataObject* pointer for the source object, which the target tests for a format it can understand.
7. *OnDragOver* returns a drop effect code, which OLE uses to set the cursor.
8. OLE calls *IDataSource::QueryContinueDrag* on the source side to find out whether the drag operation is still in progress. The MFC *COleDataSource* class responds appropriately.
9. User releases the mouse button to drop the object in the target view window.
10. OLE calls *IDropTarget::OnDrop*, which calls *OnDrop* for the target's view. Because *OnDrop* is passed a *COleDataObject* pointer, it can retrieve the desired format from that object.
11. When *OnDrop* returns in the target program, *DoDragDrop* can return in the source program.

OLE Embedded Components and Containers

- A **component** that supports **in-place activation** also supports **embedding**
- **Both in-place activation and embedding** store their data in a container's document
- **The container** can activate both.
- An in-place-capable component can run inside the container application's main window, taking over the container's menu and toolbar,
- An embedded component can run only in its own window, and that window has a special menu that does not include file commands.
- Embedding relies on two key interfaces, ***IOleObject* and *IOleClientSite,*** which are used for in-place activation as well.

Excel spreadsheet activated inside a Word document.



	A	B	C	D	E	F	G
1	Book Name	Quantity	Price ($)	Sale (%)	At Cost ($)	Supplier Email	
2	All Too Human	1	8.00	20	7.00	Catherine	
3	Don Quixote	8	10.00	0	3.50	Cindy	
4	Hamlet	3	3.50	10	15.50	David	
5	Histories	1	5.95	0	4.90	Geoff	
6	Kidnapped	1	23.45	0	17.23	Gordine	
7	Madame Bovary	10	20.48	35	9.76	James	
8	Moby Dick	2	10.35	5	14.54	Jasmine	
9	Paradise Lost	3	17.50	10	12.95	Jeff	
10	Principia Mathematica	3	4.73	0	2.98	Jennifer	
11	Spring Snow	11	22.73	40	4.97	Jessica	
12	The Brothers Karamazov	5	8.10	25	17.41	Jon	
13	The Joy of Cooking	4	10.95	5	7.35	Jon	
14	The Odyssey	4	6.98	0	11.67	Lisa	

MFC base classes—

- ***COleIPFrameWnd, COleServerDoc,* and *COleServerItem*.**
- ***COleIPFrameWnd*** class is rather like ***CFrameWnd.*** It's your

application's main frame window, which contains the view.

- It has a menu associated with it, ***IDR_SRVR_INPLACE,*** which

will be merged into the container program's menu.

- The embedded menu is ***IDR_SRVR_EMBEDDED***, and the

stand-alone menu is ***IDR_MAINFRAME***.

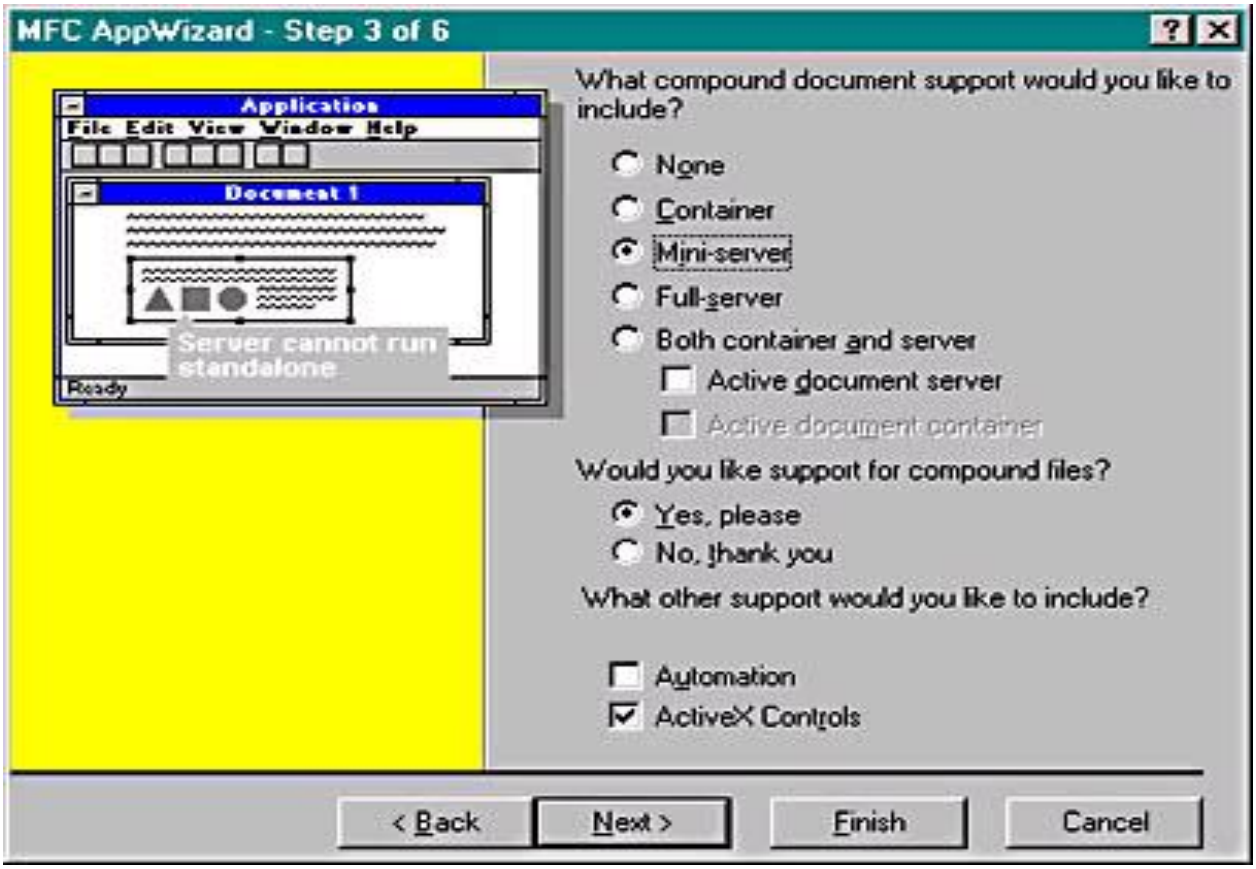
- The ***COleServerDoc*** class is a replacement for ***CDocument.***

It contains added features that support OLE connections to the container. The *C**OleServerItem*** class works with the ***COleServerDoc*** class.

The EX28A Example—An MFC In-Place-Activated Mini-Server

Here are the steps for creating the program from scratch:

1. **Run AppWizard to create the EX28A project in the \vcpp32\ex28a directory. Select Single Document interface. Click the Mini-Server option in the AppWizard Step 3 dialog shown here.**



2. **Examine the generated files.** You've got the familiar application, document, main frame, and

Header	Implementation	Class	MFC Base Class
SrvrItem.h	SrvrItem.cpp	*CEx28aSrvrItem*	*COleServerItem*
IpFrame.h	IpFrame.cpp	*CInPlaceFrame*	*COleIPFrameWnd*

view file s, but you've got two new files too.

3. **Add a text member to the document class.** Add the following public data member in the class declaration in ex28aDoc.h:

CString m_strText;

Set the string's initial value to *Initial default text* in the document's *OnNewDocument* member function.

4. **Add a dialog to modify the text.** Insert a new dialog template with an edit control, and then use ClassWizard to generate a *CTextDialog* class derived from *CDialog*.

Don't forget to include the dialog class header in ex28aDoc.cpp. Also, use ClassWizard to add a *CString* member variable named *m_strText* for the edit control.

5. **Add a new menu command in both the embedded and in-place menus.** Add a Modify menu command in both the *IDR_SRVR_EMBEDDED* and *IDR_SRVR_INPLACE* menus.

To insert this menu command on the *IDR_SRVR_EMBEDDED* menu, use the resource editor to add an EX28A-EMBED menu item on the top level, and then add a Modify option on the submenu for this item.

Next add an EX28A-INPLACE menu item on the top level of the *IDR_SRVR_INPLACE* menu and add a Modify option on the EX28A-INPLACE submenu.

To associate both Modify options with one *OnModify* function, use *ID_MODIFY* as the ID for the Modify option of both the *IDR_SRVR_EMBEDDED* and *IDR_SRVR_INPLACE* menus. Then use ClassWizard to map both Modify options to the *OnModify* function in the document class. Code the Modify command handler as shown here:

```
void CEx28aDoc::OnModify()
{ CTextDialog dlg;
  dlg.m_strText = m_strText;
   if (dlg.DoModal() == IDOK)
   { m_strText = dlg.m_strText;
    UpdateAllViews(NULL); // Trigger CEx28aView::OnDraw
     UpdateAllItems(NULL); // Trigger
                //CEx28aSrvrItem::OnDraw SetModifiedFlag();
}
```

6. **Override the view's *OnPrepareDC* function.** Use ClassWizard to generate the function, and then replace any existing code with the following line:

```
pDC->SetMapMode(MM_HIMETRIC);
```

7. **Edit the view's *OnDraw* function. The following code in ex28aView.cpp draws a 2-cm circle centered in the client rectangle, with the text wordwrapped in the window:**

```
void CEx28aView::OnDraw(CDC* pDC)
{
        CEx28aDoc* pDoc = GetDocument();  ASSERT_VALID(pDoc);
        CFont font; font.CreateFont(-500, 0, 0, 0, 400, FALSE, FALSE, 0,
ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
    DEFAULT_QUALITY, DEFAULT_PITCH | FF_SWISS, "Arial");
CFont* pFont = pDC->SelectObject(&font);
CRect rectClient; GetClientRect(rectClient);
CSize sizeClient = rectClient.Size();
pDC->DPtoHIMETRIC(&sizeClient);
CRect rectEllipse(sizeClient.cx / 2 - 1000, -sizeClient.cy / 2 + 1000, sizeClient.cx / 2 + 1000, -
sizeClient.cy / 2 - 1000);
 pDC->Ellipse(rectEllipse);
pDC->TextOut(0, 0, pDoc->m_strText);
pDC->SelectObject(pFont);
}
```

8. Edit the server item's *OnDraw* function. The following code in the SrvrItem.cpp file tries to draw the same circle drawn in the view's *OnDraw* function.

```
BOOL CEx28aSrvrItem::OnDraw(CDC* pDC, CSize& rSize)
{ // Remove this if you use rSize
UNREFERENCED_PARAMETER(rSize);
CEx28aDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc); // TODO: set mapping mode and extent // (The extent is usually the same as the
size returned from // OnGetExtent)
pDC->SetMapMode(MM_ANISOTROPIC);
 pDC->SetWindowOrg(0,0);
 pDC->SetWindowExt(3000, -3000);
CFont font; font.CreateFont(-500, 0, 0, 0, 400, FALSE, FALSE, 0, ANSI_CHARSET,
OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
DEFAULT_PITCH | FF_SWISS, "Arial");
CFont* pFont = pDC->SelectObject(&font);
CRect rectEllipse(CRect(500, -500, 2500, -2500));
```

```
pDC->Ellipse(rectEllipse);
pDC->TextOut(0, 0, pDoc->m_strText);
 pDC->SelectObject(pFont); return TRUE;
}
```

9. Edit the document's *Serialize* function. The framework takes care of loading and saving the document's data from and to an OLE stream named *Contents* that is attached to the object's main storage. You simply write normal serialization code, as shown here:

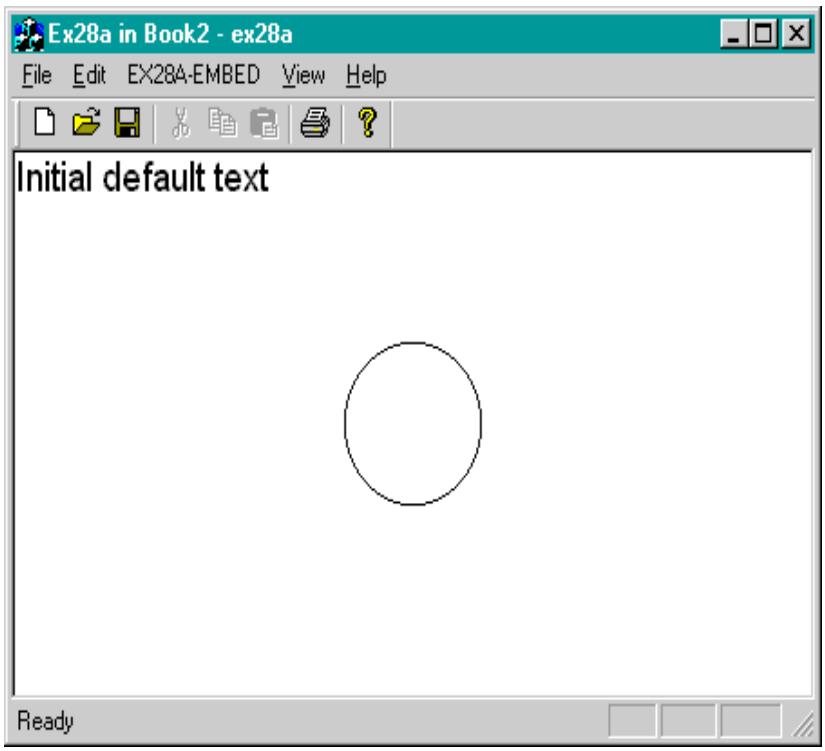
```
void CEx28aDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  { ar << m_strText; }
  else
 { ar >> m_strText; }
 }
```

There is also a *CEx28aSrvrItem::Serialize* function that delegates to the document *Serialize* function.

10. Build and register the EX28A application. You must run the application directly once to update the Registry.

11. Test the EX28A application. You need a container program that supports in-place activation. Use Microsoft Excel 97 or a later version if you have it, or build the project in the MFC DRAWCLI sample. Choose the container's Insert Object menu item. If this option does not appear on the Insert menu, it might appear on the Edit menu instead. Then select Ex28a Document from the list.

When you first insert the EX28A object, you'll see a hatched border, which indicates that the object is in-place active. The bounding rectangle is 3-by-3-cm square, with a 2-cm circle in the center, as illustrated here.



ActiveX Controls

Introduction

- Microsoft Visual Basic (VB) was introduced in 1991 and has proven to be a wildly popular and successful application development system for Microsoft Windows
- The 16-bit versions of VB supported Visual Basic controls (VBXs), ready-to-run software components that VB developers could buy or write themselves.

- VBXs became the center of a whole industry, and pretty soon there were hundreds of them.
- The VBX standard, which was highly dependent on the 16-bit segment architecture, did not make it to the 32-bit world.
- ActiveX Controls (formerly known as OLE controls, or OCXs) are the industrial-strength replacement for VBXs based on Microsoft COM technology.
- ActiveX controls can be used by application developers in both VB and Visual C++ 6.0.
- ActiveX controls can be written in C++ with the help of the MFC library or with the help of the ActiveX Template Library (ATL).

ActiveX Controls vs. Ordinary Windows Controls

Ordinary Controls—A Frame of Reference

- These controls are all child windows that you use most often in dialogs, and they are represented by MFC classes such as *CEdit* and *CTreeCtrl*. The client program is always responsible for the creation of the control's child window.
- Ordinary controls send notification command messages (standard Windows messages), such as BN_CLICKED, to the dialog.
- If you want to perform an action on the control, you call a C++ control class member function, which sends a Windows message to the control.
- The controls are all windows in their own right. All the MFC control classes are derived from *CWnd*, so if you want to get the text from an edit control, you call *CWnd::GetWindowText*. But even that function works by sending a message to the control.
- Windows controls are an integral part of Windows, even though the Windows common controls are in a separate DLL.

How ActiveX Controls Are Similar to Ordinary Controls

- You can consider an ActiveX control to be a child window, just as an ordinary control is.
- If you want to include an ActiveX control in a dialog, you use the dialog editor to place it there, and the identifier for the control turns up in the resource template.
- If you're creating an ActiveX control on the fly, you call a *Create* member function for a class that represents the control, usually in the WM_CREATE handler for the parent window.
- When you want to manipulate an ActiveX control, you call a C++ member function, just as you do for a Windows control. The window that contains a control is called a container.

ActiveX Controls vs. Ordinary Windows Controls

How ActiveX Controls Are Different from Ordinary Controls—Properties and Methods

- The most prominent ActiveX Controls features are properties and methods.
- Properties have symbolic names that are matched to integer indexes. For each property, the control designer assigns a property name, such as BackColor or GridCellEffect, and a property type, such as string, integer, or double.
- The client program can set an individual ActiveX control property by specifying the property's integer index and its value. The client can get a property by specifying the index and accepting the appropriate return value.
- In certain cases, ClassWizard lets you define data members in your client window class that are associated with the properties of the controls the client class contains. The generated Dialog Data Exchange (DDX) code exchanges data between the control properties and the client class data members.

ActiveX Controls vs. Ordinary Windows Controls

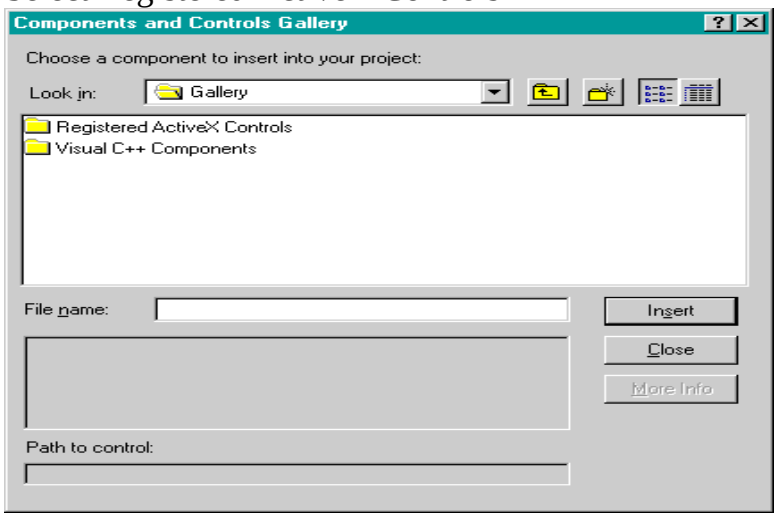
- **How ActiveX Controls Are Different from Ordinary Controls—Properties and Methods**
- **ActiveX Controls methods are like functions. A method has a symbolic name, a set of parameters, and a return value. You call a method by calling a C++ member function of the class that represents the control. A control designer can define any needed methods, such as PreviousYear, LowerControlRods, and so forth.**

- **An ActiveX control doesn't send WM_ notification messages to its container the way ordinary controls do; instead, it "fires events."**
- **An event has a symbolic name and can have an arbitrary sequence of parameters—it's really a container function that the control calls.**
- **Like ordinary control notification messages, events don't return a value to the ActiveX control.**
- **Examples of events are Click, KeyDown, and NewMonth. Events are mapped in your client class just as control notification messages are.**
- **ActiveX Controls vs. Ordinary Windows Controls**
- **How ActiveX Controls Are Different from Ordinary Controls—Properties and Methods**
- **In the MFC world, ActiveX controls act just like child windows, but there's a significant layer of code between the container window and the control window.**
- **In fact, the control might not even have a window. When you call Create, the control's window isn't created directly; instead, the control code is loaded and given the command for "in-place activation." The ActiveX control then creates its own window, which MFC lets you access through a CWnd pointer. It's not a good idea for the client to use the control's hWnd directly, however.**
- **A DLL is used to store one or more ActiveX controls, but the DLL often has an OCX filename extension instead of a DLL extension.**
- **Your container program loads the DLLs when it needs them, using sophisticated COM techniques that rely on the Windows Registry.**
- **Once you specify an ActiveX control at design time, it will be loaded for you at runtime.**
- **Obviously, when you ship a program that requires special ActiveX controls, you'll have to include the OCX files and an appropriate setup program.**

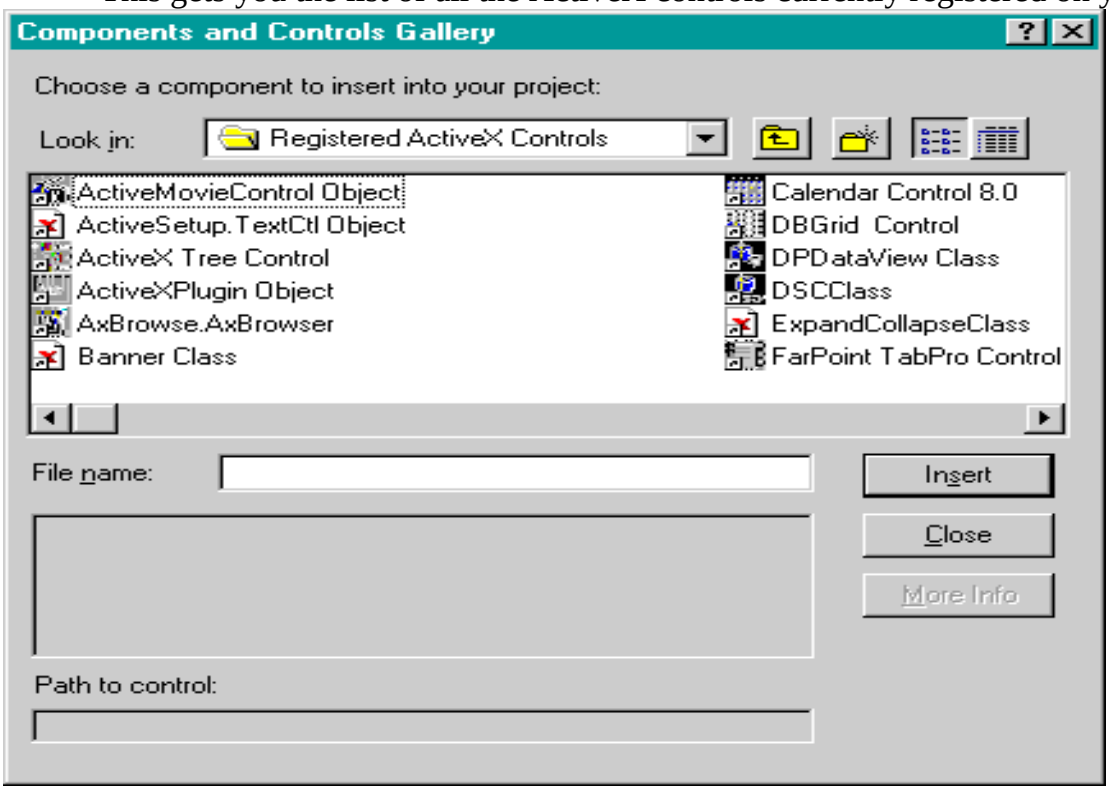
Installing ActiveX Controls

- Your first step is to copy the ActiveX control's DLL to \Windows\System for Microsoft Windows 95 or \Winnt\System32 for Microsoft Windows NT. Copy associated files such as help (HLP) or license (LIC) files to the same directory.
- Your next step is to register the control in the Windows Registry. Actually, the ActiveX control registers itself when a client program calls a special exported function. The Windows utility Regsvr32 is a client that accepts the control name on the command line.
- After you register your ActiveX control, you must install it in each project that uses it. That doesn't mean that the OCX file gets copied. It means that ClassWizard generates a copy of a C++ class that's specific to the control, and it means that the control shows up in the dialog editor control palette for that project.

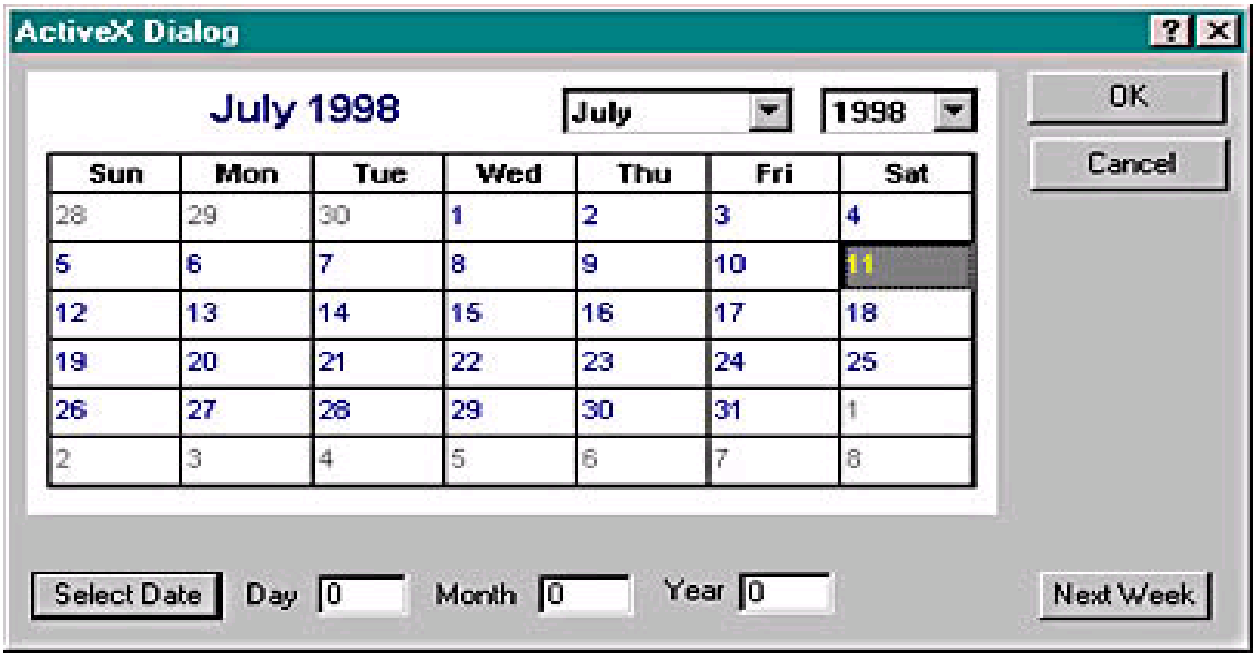
Installing ActiveX Controls

- Choose Add To Project from the Project menu and then choose Components And Controls.
- Select Registered ActiveX Controls
- 

This gets you the list of all the ActiveX controls currently registered on your system.



The Calendar Control



Properties	Methods	Events
BackColor	AboutBox	AfterUpdate
Day	NextDay	BeforeUpdate
DayFont	NextMonth	Click
DayFontColor	NextWeek	DblClick
DayLength	NextYear	KeyDown

FirstDay	PreviousDay	KeyPress
GridCellEffect	PreviousMonth	KeyUp
GridFont	PreviousWeek	NewMonth
GridFontColor	PreviousYear	NewYear
GridLinesColor	Refresh	
Month	Today	
MonthLength		
ShowDateSelectors		

- Each of the properties, methods, and events has a corresponding integer identifier.
- Information about the names, types, parameter sequences, and integer IDs is stored inside the control and is accessible to ClassWizard at container design time.

ActiveX Control Container Programming

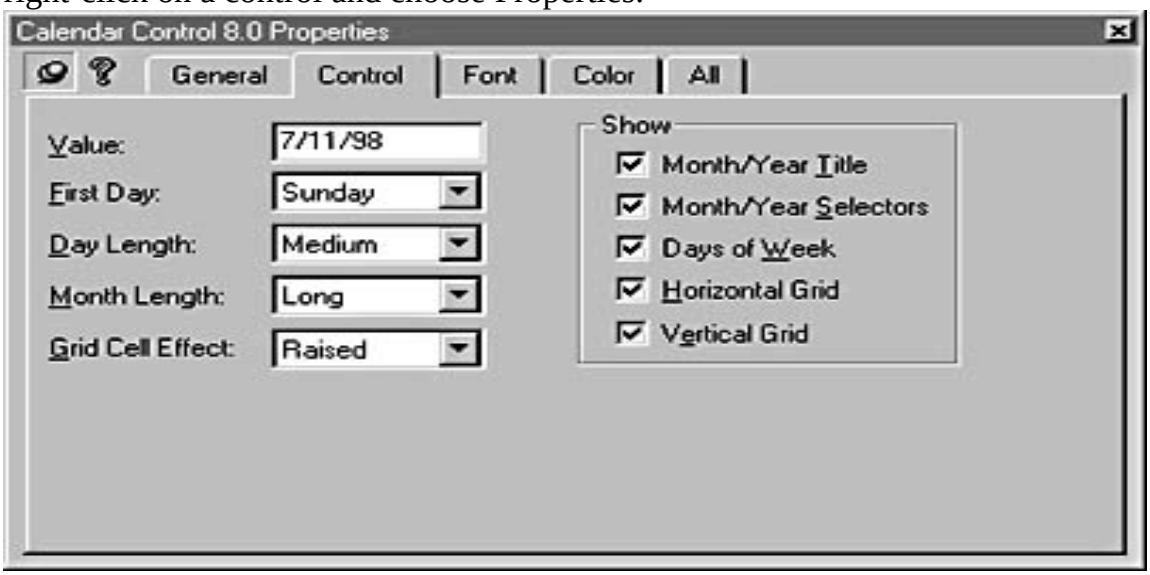
MFC and ClassWizard support ActiveX controls both in dialogs and as "child windows."

To use ActiveX controls, you must understand how a control grants access to properties, and you must understand the interactions between your Dialog Data Exchange (DDX) code and those property values.

Property Access

The ActiveX control developer designates certain properties for access at design time.

Those properties are specified in the property pages that the control displays in the dialog editor when you right-click on a control and choose Properties.

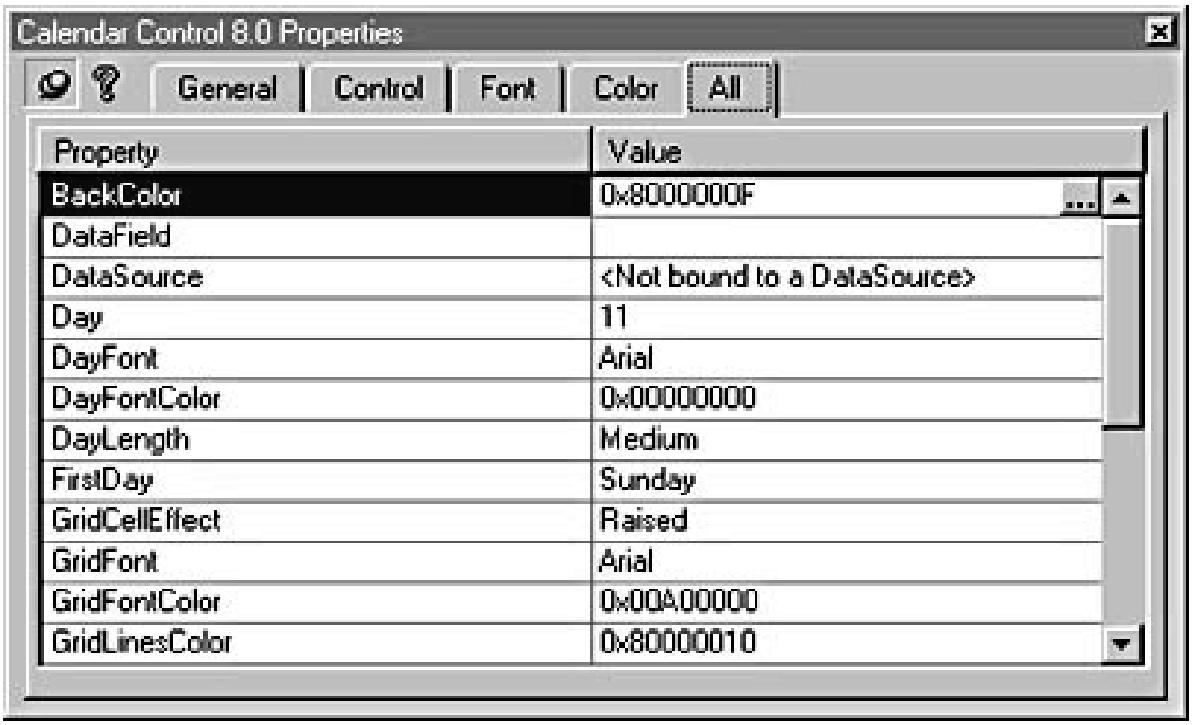


Property Access

When you click on the All tab, you will see a list of all the design- time-accessible properties, which might include a few properties not found on the Control tab

All the control's properties, including the design-time properties, are accessible at runtime.

Some properties, however, might be designated as read-only.



ClassWizard's C++ Wrapper Classes for ActiveX Controls

- When you insert an ActiveX control into a project, ClassWizard generates a C++ wrapper class, derived from *CWnd*, that is tailored to your control's methods and properties.
- The class has member functions for all properties and methods, and it has constructors that you can use to dynamically create an instance of the control.

ClassWizard's C++ Wrapper Classes for ActiveX Controls

```
unsigned long CCalendar::GetBackColor()
{
   unsigned long result;
   InvokeHelper(DISPID_BACKCOLOR, DISPATCH_PROPERTYGET,                VT_I4,
(void*)&result, NULL);
   return result;
}
void CCalendar::SetBackColor(unsigned long newValue)
{
   static BYTE parms[] =       VTS_I4;
   InvokeHelper(DISPID_BACKCOLOR, DISPATCH_PROPERTYPUT,
         VT_EMPTY, NULL, parms, newValue);
}
```

ClassWizard's C++ Wrapper Classes for ActiveX Controls

```
short CCalendar::GetDay()
{
        short result;
        InvokeHelper(0x11, DISPATCH_PROPERTYGET, VT_I2,          (void*)&result,
NULL);
     return result;
}
void CCalendar::SetDay(short nNewValue)
{
     static BYTE parms[] =       VTS_I2;

     InvokeHelper(0x11, DISPATCH_PROPERTYPUT, VT_EMPTY,NULL, parms, nNewValue);
}
```

ClassWizard's C++ Wrapper Classes for ActiveX Controls

```
COleFont CCalendar::GetDayFont()
{
    LPDISPATCH pDispatch;
    InvokeHelper(0x1, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&pDispatch,
NULL);
    return COleFont(pDispatch);
}
void CCalendar::SetDayFont(LPDISPATCH newValue)
{
    static BYTE parms[] =       VTS_DISPATCH;
    InvokeHelper(0x1, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
newValue);
}
```

ClassWizard's C++ Wrapper Classes for ActiveX Controls

```
VARIANT CCalendar::GetValue()
{
        VARIANT result;
        InvokeHelper(0xc, DISPATCH_PROPERTYGET, VT_VARIANT, (void*)&result,
NULL);
        return result;
}
void CCalendar::SetValue(const VARIANT& newValue)
{
        static BYTE parms[] =       VTS_VARIANT;
        InvokeHelper(0xc, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
&newValue);
}
```

ClassWizard's C++ Wrapper Classes for ActiveX Controls

```
void CCalendar::NextDay()
{
    InvokeHelper(0x16, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}
void CCalendar::NextMonth()
{
    InvokeHelper(0x17, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}
```

ClassWizard's C++ Wrapper Classes for ActiveX Controls

- The first parameter of each *InvokeHelper* function match with the dispatch ID for the corresponding property or method in the Calendar control property list.
- Properties always have separate *Set* and *Get* functions.
- To call a method, simply call the corresponding function.
- To call the NextDay method from a dialog class function, you write code such as this:
- m_calendar.NextDay();
- *m_calendar* is an object of class *CCalendar*, the wrapper class for the Calendar control.

AppWizard Support for ActiveX Controls

- When the AppWizard ActiveX Controls option is checked (the default), AppWizard inserts the following line in your application class *InitInstance* member function:
- AfxEnableControlContainer();
- It also inserts the following line in the project's StdAfx.h file:

#include <afxdisp.h>

If you decide to add ActiveX controls to an existing project that doesn't include the two lines above, you can simply add the lines.

ClassWizard and the Container Dialog

If your template contains one or more ActiveX controls, you can use ClassWizard to add data members and event handler functions.

Dialog Class Data Members vs. Wrapper Class Usage

- What kind of data members can you add to the dialog for an ActiveX control?
- If you want to set a control property before you call *DoModal* for the dialog, you can add a dialog data member for that property.
- If you want to change properties inside the dialog member functions, you must take another approach: you add a data member that is an object of the wrapper class for the ActiveX control.

Dialog Class Data Members vs. Wrapper Class Usage

- MFC DDX logic.
- The *CDialog::OnInitDialog* function calls *CWnd::UpdateData(FALSE)* to read the dialog class data members, and the *CDialog::OnOK* function calls *UpdateData(TRUE)* to write the members.
- Suppose you added a data member for each ActiveX control property and you needed to get the Value property value in a button handler.
- If you called *UpdateData(FALSE)* in the button handler, it would read all the property values from all the dialog's controls—clearly a waste of time.
- It's more effective to avoid using a data member and to call the wrapper class *Get* function instead.
- To call that function, you must first tell ClassWizard to add a wrapper class object data member.

Dialog Class Data Members vs. Wrapper Class Usage

- Suppose you have a Calendar wrapper class *CCalendar* and you have an *m_calendar* data member in your dialog class. If you want to get the Value property, you do it like this:
    - COleVariant var = m_calendar.GetValue();
- you want to set the day to the 5th of the month before the control is displayed. To do this by hand, add a dialog class data member *m_sCalDay* that corresponds to the control's short integer Day property. Then add the following line to the *DoDataExchange* function:
- DDX_OCShort(pDX, ID_CALENDAR1, 0x11, m_sCalDay);
- Following codes construct and display dialog
  CMyDialog dlg;
  dlg.m_sCalDay = 5;
  dlg.DoModal();

Dialog Class Data Members vs. Wrapper Class Usage

- The DDX code takes care of setting the property value from the data member before the control is displayed.
- No other programming is needed.

Mapping ActiveX Control Events

- ClassWizard lets you map ActiveX control events the same way you map Windows messages and command messages from controls.
- If a dialog class contains one or more ActiveX controls, ClassWizard adds and maintains an event sink map that connects mapped events to their handler functions.
- It works something like a message map.

Locking ActiveX Controls in Memory

- Normally, an ActiveX control remains mapped in your process as long as its parent dialog is active.
- That means it must be reloaded each time the user opens a modal dialog.
- The reloads are usually quicker than the initial load because of disk caching, but you can lock the control into memory for better performance.
- To do so, add the following line in the overridden *OnInitDialog* function after the base class call:

- AfxOleLockControl(m_calendar.GetClsid());
- The ActiveX control remains mapped until your program exits or until you call the *AfxOleUnlockControl* function.

Example—An ActiveX Control Dialog Container

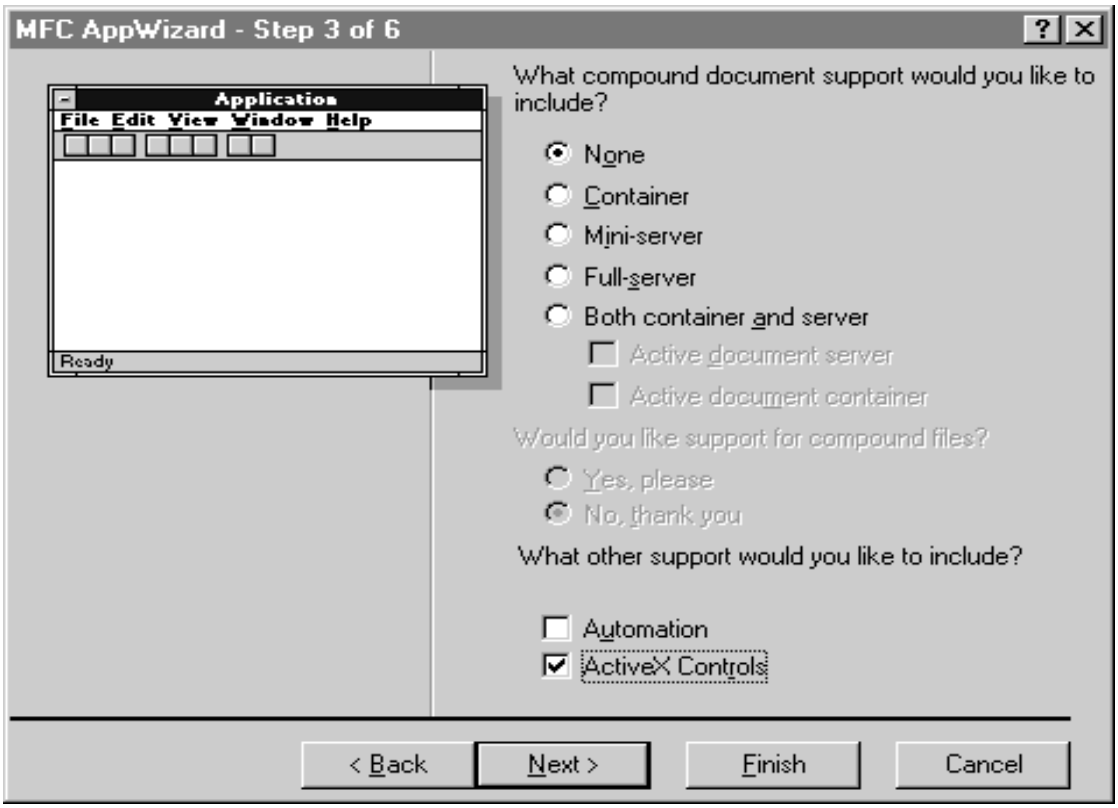
Step 1:

Verify that the Calendar control is registered. If the control does not appear in the Visual C++ Gallery's Registered ActiveX Controls page, copy the files MSCal.ocx, MSCal.hlp, and MSCal.cnt to your system directory and register the control by running the REGCOMP program.

Step 2:

Run AppWizard to produce \vcpp32\ex08a\ex08a.

Accept all of the default settings but two: select Single Document and deselect Printing And Print Preview.

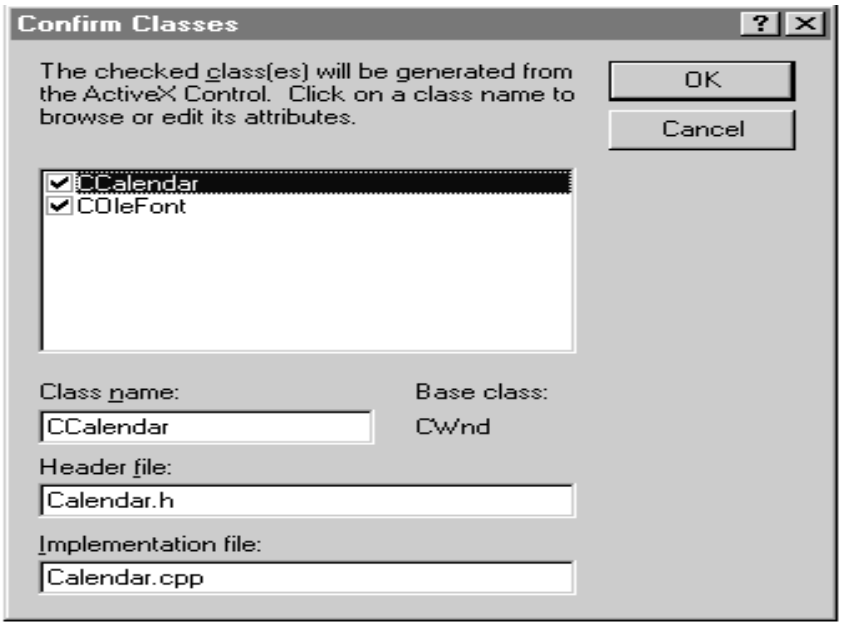


In the AppWizard Step 3 dialog, make sure the ActiveX Controls option is selected, as shown below.

Step 3: Install the Calendar control in the EX08A project.

Choose Add To Project from Visual C++'s Project menu, and then choose Components And Controls. Choose Registered ActiveX Controls, and then choose Calendar Control 8.0.

ClassWizard generates two classes in the EX08A directory



Step 4: Edit the Calendar control class to handle help messages.

Add Calendar.cpp to the following message map code:

```
BEGIN_MESSAGE_MAP(CCalendar, CWnd)      ON_WM_HELPINFO()
END_MESSAGE_MAP()
```

In the same file, add the *OnHelpInfo* function:

```
BOOL CCalendar::OnHelpInfo(HELPINFO* pHelpInfo)
{
// Edit the following string for your system     ::WinHelp(GetSafeHwnd(),
"c:\\winnt\\system32\\mscal.hlp",       HELP_FINDER, 0);
return FALSE;
 }
```

In Calendar.h, add the function prototype and declare the message map:

```
protected:
        afx_msg BOOL OnHelpInfo(HELPINFO* pHelpInfo);
DECLARE_MESSAGE_MAP()
```

The *OnHelpInfo* function is called if the user presses the F1 key when the Calendar control has the input focus.

We have to add the message map code by hand because ClassWizard doesn't modify generated ActiveX classes.

Step 5:Use the dialog editor to create a new dialog resource.

Choose Resource from Visual C++'s Insert menu, and then choose Dialog.

The dialog editor assigns the ID *IDD_DIALOG1* to the new dialog.

Next change the ID to *IDD_ACTIVEXDIALOG*, change the dialog caption to *ActiveX Dialog*, and set the dialog's Context Help property (on the More Styles page).

Accept the default OK and Cancel buttons with the IDs *IDOK* and *IDCANCEL*, and then add the other controls .

Step 5: Use the dialog editor to create a new dialog resource.(2)

- Make the Select Date button the default button. Drag the Calendar control from the control palette.
- Then set an appropriate tab order. Assign control IDs as shown in the following table.

Control ID

Calendar control	*IDC_CALENDAR1*
Select Date button	*IDC_SELECTDATE*
Edit control	*IDC_DAY*
Edit control	*IDC_MONTH*
Edit control	*IDC_YEAR*
Next Week button	*IDC_NEXTWEEK*

Step 6: Use ClassWizard to create the *CActiveXDialog* class

If you run ClassWizard directly from the dialog editor window, it will know that you want to create a *CDialog*-derived class based on the *IDD_ACTIVEXDIALOG* template.

Simply accept the default options, and name the class *CActiveXDialog*.

Click on the ClassWizard Message Maps tab, and then add the message handler functions .

To add a message handler function, click on an object ID, click on a message, and click the Add Function button.

If the Add Member Function dialog box appears, type the function name and click the OK button.

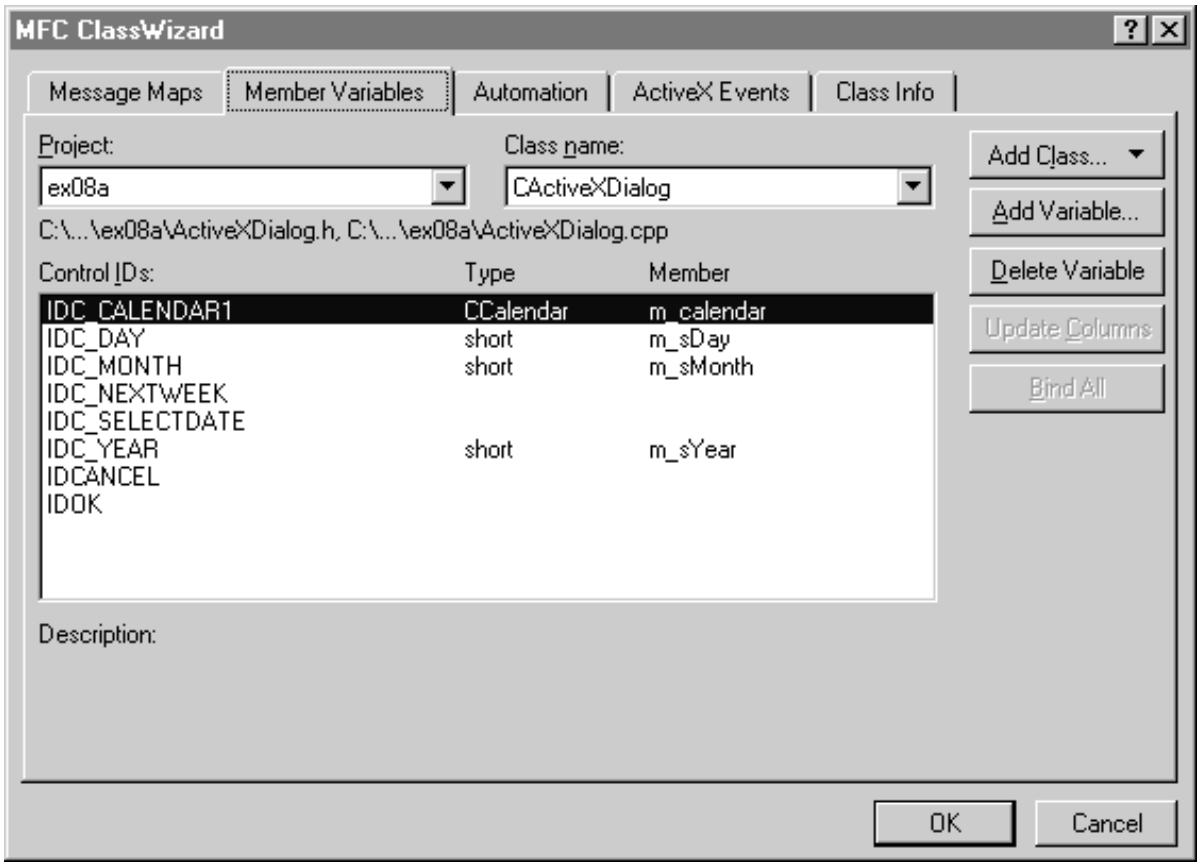
Step 6: Use ClassWizard to create the *CActiveXDialog* class (2)

Message Handler Function

Object ID	**Message**	**Member Function**
CActiveXDialog	WM_INITDIALOG	*OnInitDialog* (virtual function)
IDC_CALENDAR1	NewMonth (event)	*OnNewMonthCalendar1*
IDC_SELECTDATE	BN_CLICKED	*OnSelectDate*
IDC_NEXTWEEK	BN_CLICKED	*OnNextWeek*
IDOK	BN_CLICKED	*OnOK* (virtual function)

Step 7:Use ClassWizard to add data members to the *CActiveXDialog* class.

Click on the Member Variables tab, and then add the data members



Step 8: Edit the *CActiveXDialog* class.

Add the *m_varValue* and *m_BackColor* data members, and then edit the code for the five handler functions *OnInitDialog*, *OnNewMonthCalendar1*, *OnSelectDate*, *OnNextWeek*, and *OnOK*.

Step 8.1 : Edit the *CActiveXDialog* class.

```
void CActiveXDialog::OnNewMonthCalendar1()
{
AfxMessageBox("EVENT: CActiveXDialog::OnNewMonthCalendar1");
 }
void CActiveXDialog::OnSelectDate()
{   CDataExchange dx(this, TRUE);
    DDX_Text(&dx, IDC_DAY, m_sDay);
    DDX_Text(&dx, IDC_MONTH, m_sMonth);
    DDX_Text(&dx, IDC_YEAR, m_sYear);
    m_calendar.SetDay(m_sDay);
    m_calendar.SetMonth(m_sMonth);
    m_calendar.SetYear(m_sYear);
}
void CActiveXDialog::OnNextWeek()
{
 m_calendar.NextWeek();
}
void CActiveXDialog::OnOK()
{
 CDialog::OnOK(); m_varValue = m_calendar.GetValue();
 // no DDX for VARIANTs
}
```

Step 8.2 : Edit the *CActiveXDialog* class. (3)

- The *OnSelectDate* function is called when the user clicks the Select Date button.

- The function gets the day, month, and year values from the three edit controls and transfers them to the control's properties. ClassWizard can't add DDX code for the BackColor property, so you must add it by hand.
- In addition, there's no DDX code for *VARIANT* types, so you must add code to the *OnInitDialog* and *OnOK* functions to set and retrieve the date with the control's Value property.

Step 9 : Connect the dialog to the view.
Use ClassWizard to map the WM_LBUTTONDOWN message, and then edit the handler function as follows:

```
void CEx08aView::OnLButtonDown(UINT nFlags, CPoint point)
{
        CActiveXDialog dlg;
        dlg.m_BackColor = RGB(255, 251, 240); // light yellow COleDateTime today =
COleDateTime::GetCurrentTime();
dlg.m_varValue= COleDateTime(today.GetYear(), today.GetMonth(), today.GetDay(), 0, 0, 0);
        if (dlg.DoModal() == IDOK)
        {
        COleDateTime date(dlg.m_varValue);      AfxMessageBox(date.Format("%B %d,
%Y"));
        }
 }
```

The code sets the background color to light yellow and the date to today's date, displays the modal dialog, and reports the date returned by the Calendar control. You'll need to include ActiveXDialog.h in ex08aView.cpp.

Step 10: Edit the virtual *OnDraw* function in the file ex08aView.cpp.
To prompt the user to press the left mouse button, replace the code in the view class *OnDraw* function with this single line:

```
pDC->TextOut(0, 0, "Press the left mouse button here.");
```

Step 11: Build and test the EX08A application.

- Open the dialog, enter a date in the three edit controls, and then click the Select Date button.
- Click the Next Week button.
- Try moving the selected date directly to a new month, and observe the message box that is triggered by the NewMonth event.
- Watch for the final date in another message box when you click OK.
- Press the F1 key for help on the Calendar control.

Creating ActiveX Controls at Runtime

- Insert the component into your project. ClassWizard will create the files for a wrapper class.
- Add an embedded ActiveX control wrapper class data member to your dialog class or other C++ window class. An embedded C++ object is then constructed and destroyed along with the window object.
- Choose Resource Symbols from Visual C++'s View menu. Add an ID constant for the new control.
- If the parent window is a dialog, use ClassWizard to map the dialog's WM_INITDIALOG message, thus overriding *CDialog-::OnInitDialog*. For other windows, use ClassWizard to map the WM_CREATE message. The new function should call the embedded control class's *Create* member function. This call indirectly displays the new control in the dialog. The control will be properly destroyed when the parent window is destroyed.
- In the parent window class, manually add the necessary event message handlers and prototypes for your new control. Don't forget to add the event sink map macros.

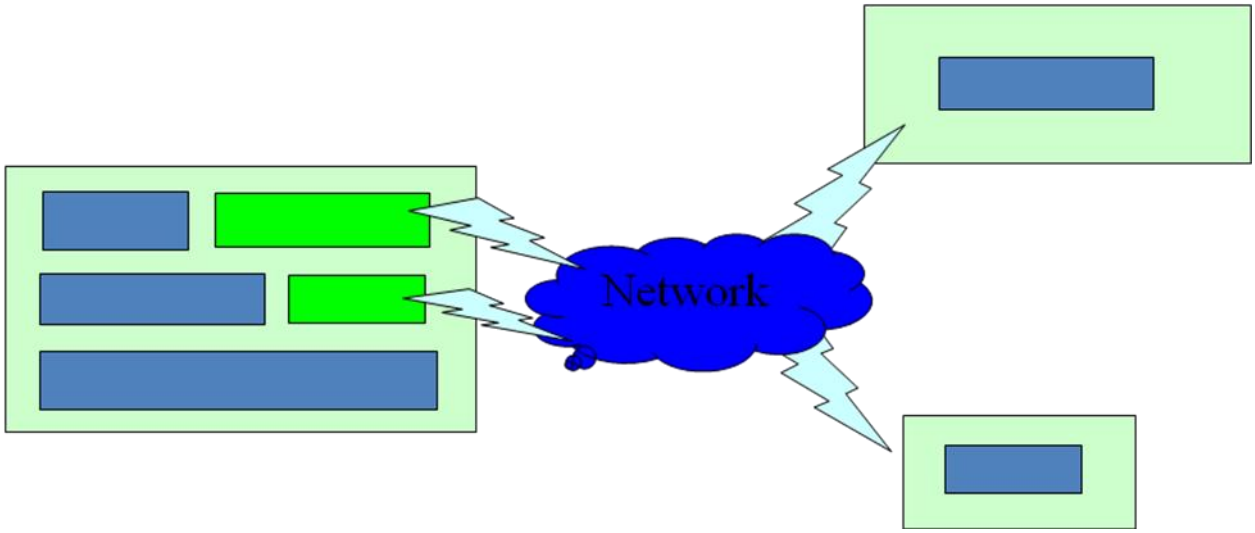
- ClassWizard doesn't help you with event sink maps when you add a dynamic ActiveX control to a project.
- Consider inserting the target control in a dialog in another temporary project.
- After you're finished mapping events, simply copy the event sink map code to the parent window class in your main project.

Component Object Model

- COM is a powerful integrating technology.

It allows developers to write software that runs regardless of issues such as thread awareness and language choice

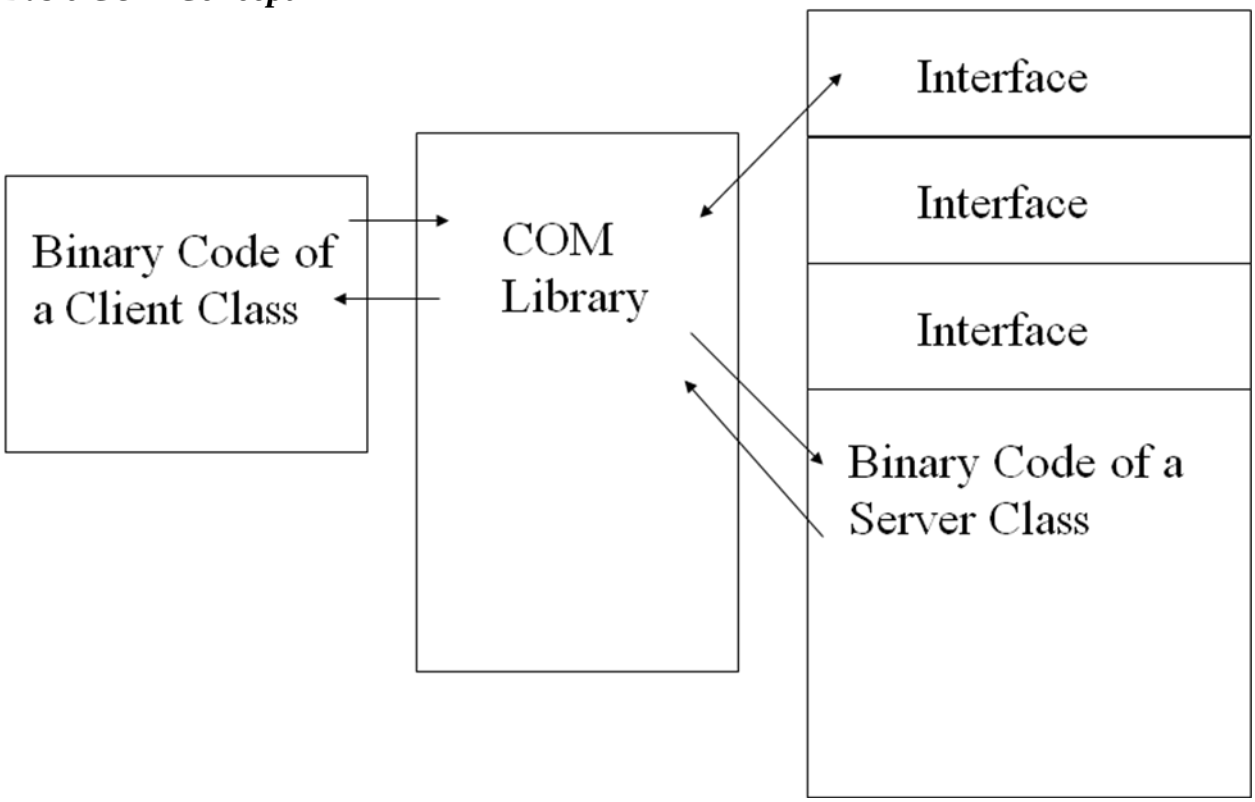
Benefits of COM – Distributed Components



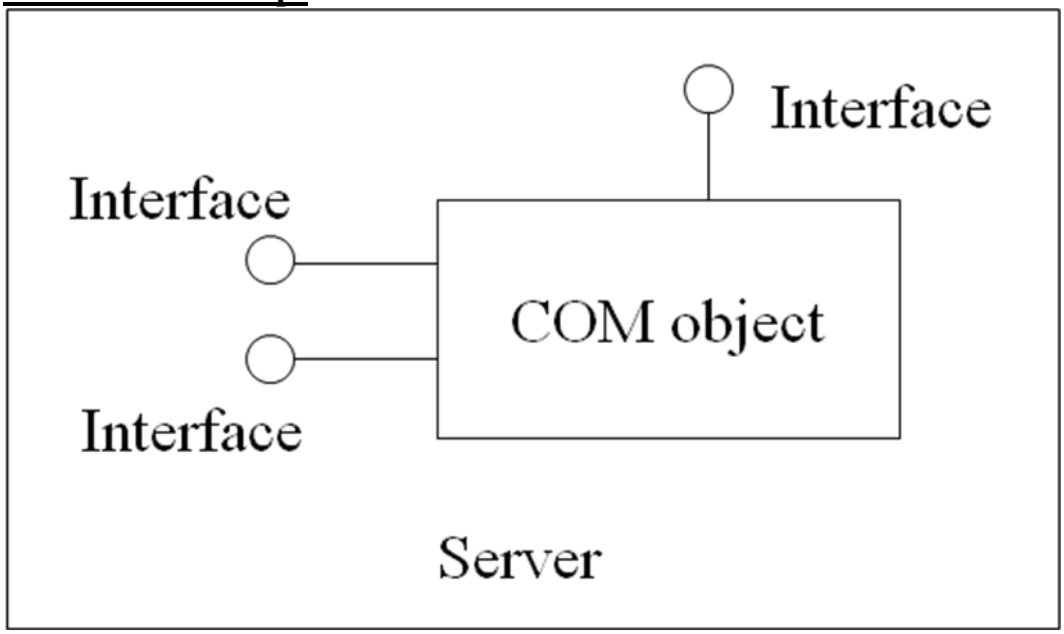
Component Object Model

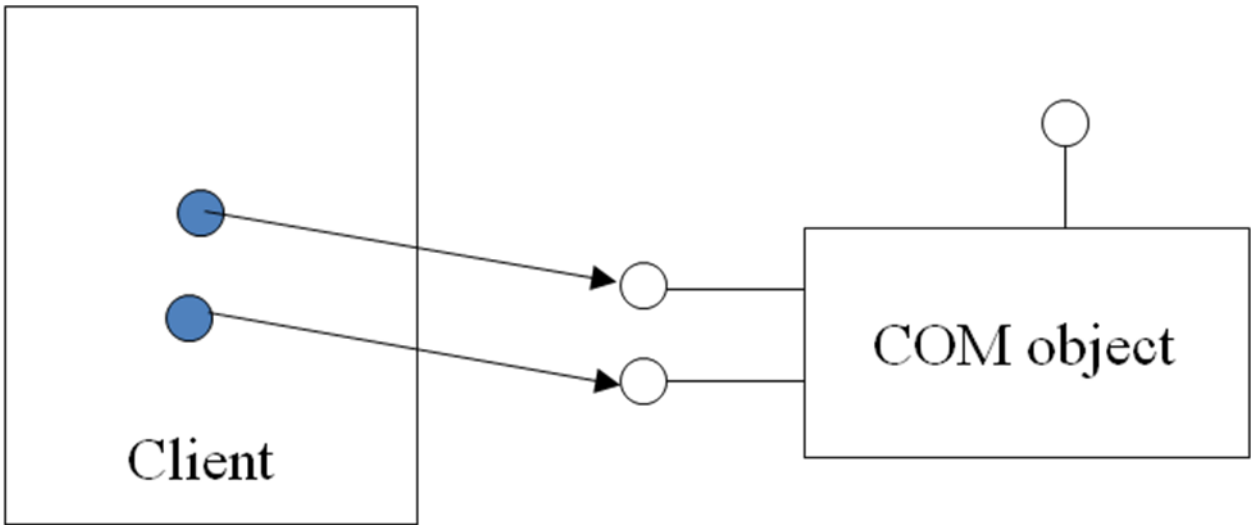
- How should one chunk of software access the services provided by another chunk of software?
- COM: A standard approach to access all kinds of software services, regardless of how they are provided
- COM is not a programming language
- COM is not DLL
- COM is not only a set of API or functions
- with the benefits of object orientation
- language independent
    - COM defines a binary interface that objects must support
- with simple and efficient.
- available on Windows, Windows NT.

Basic COM Concept



Basic COM Concept





Identifying an Interface

- Human-readable name
- Globally Unique Identifier (GUID)
- Interface Identifier (IID)

Interface Definition Language

- **uuid(E3BE7D4D-F26C-4C35-B694-ABA329A4A0E5),**
- **version(1.0),**
- **helpstring("aks_ATL 1.0 Type Library")**

Immutability of the Interfaces

- Once an interface has been implemented in released software, it cannot be changed
- To add new functionality or to modify existing functionality requires defining an entirely new interface, with a new and different IID
- The creator of the software is free to stop supporting the original interface but is absolutely prohibited from changing it

Changing Features to an interface

- The object's creator must define a new interface, say "multiply" that includes the new or changed methods and the COM object continues to support "add" as before, but it now also support "multiply".
- Clients that are unaware of the upgrade never ask for a pointer to "mutiply", they continue to use "add" as before

COM Classes

- Class identifier (CLSID)
- An object of a specific class supports a certain set of interfaces
- An object's class identifies a particular implementation of a group of interfaces

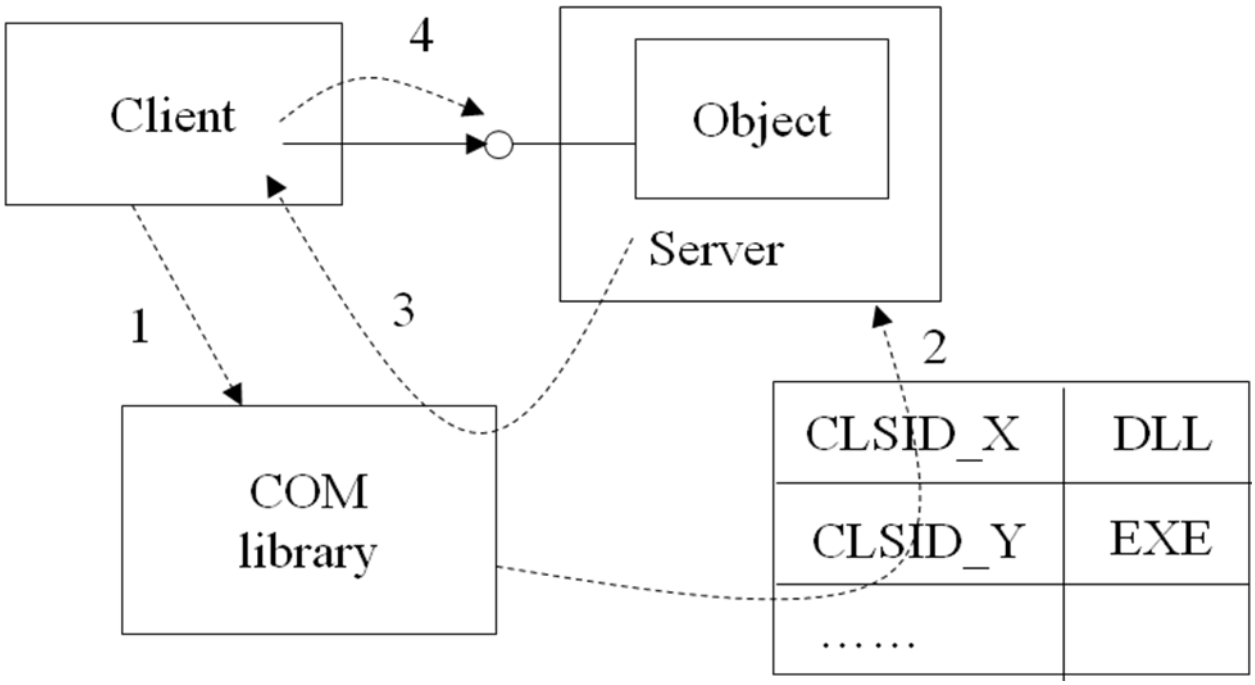
COM Library

- The COM library implements a group of functions that supply basic services to objects and their clients
- The COM library's services are accessed through ordinary function calls

System Registry

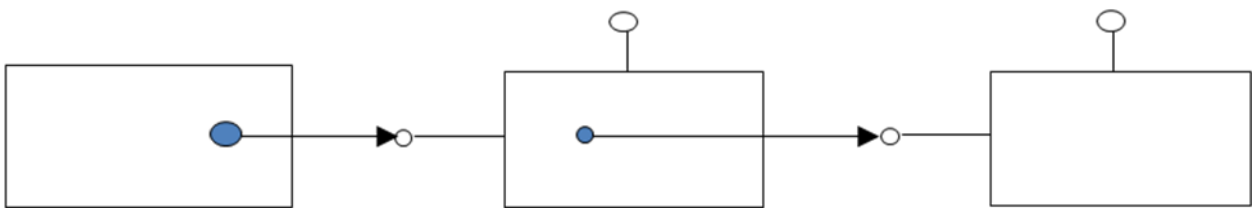
- The classes of all objects that the COM library will be asked to create on this machine must be registered
- Registry mapping includes
  - CLSID
  - Kinds of servers
  - Pathname for the file containing the server's DLL or executable, or for where to find remote server's executable

Creating a Single Object

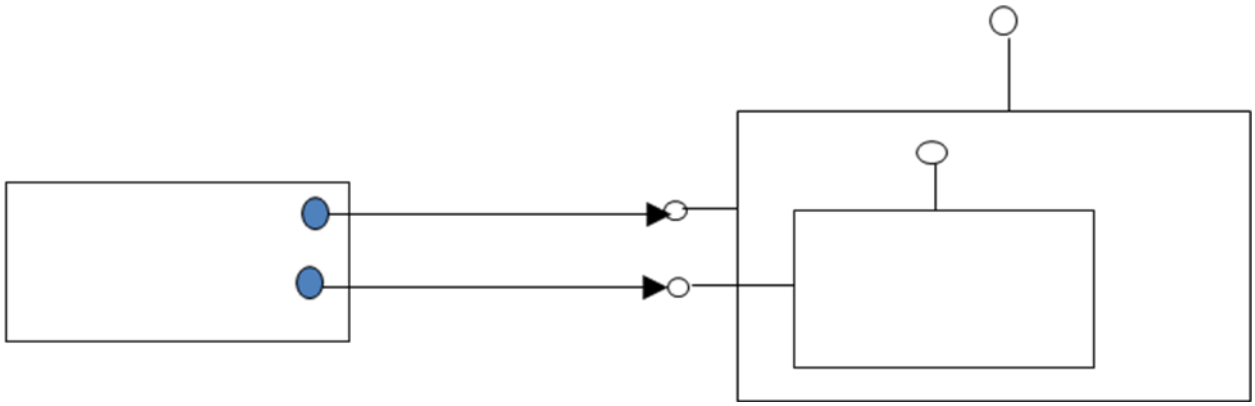


Reusing COM Objects

- One COM object can't reuse another's code through inheritance
- Containment (delegation)



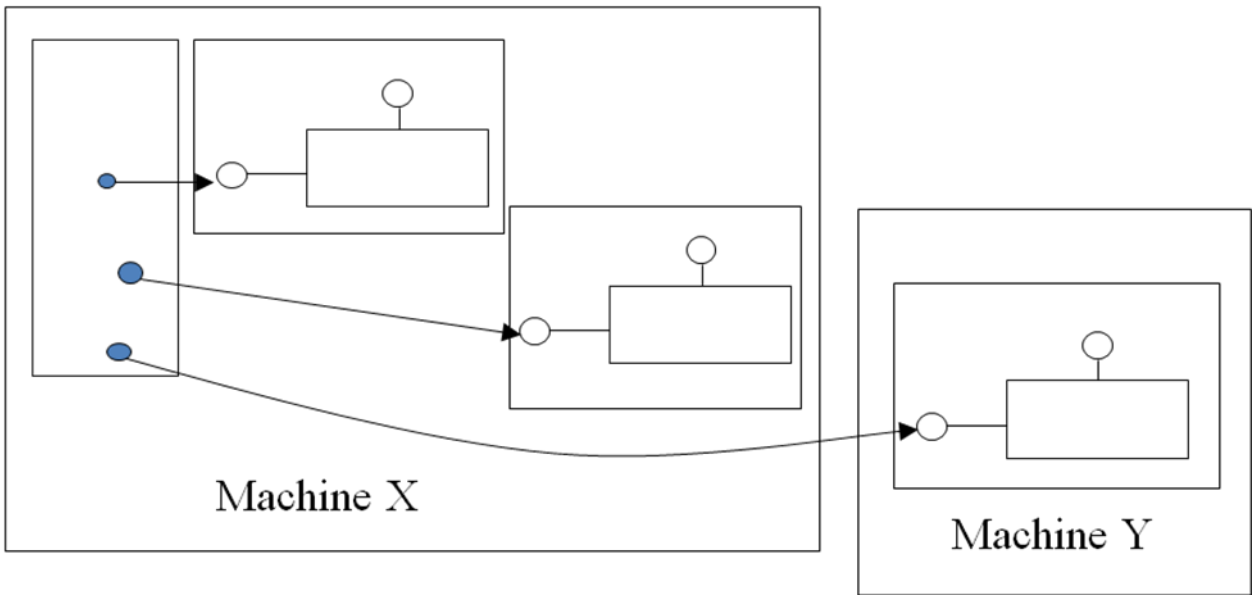
Aggregation



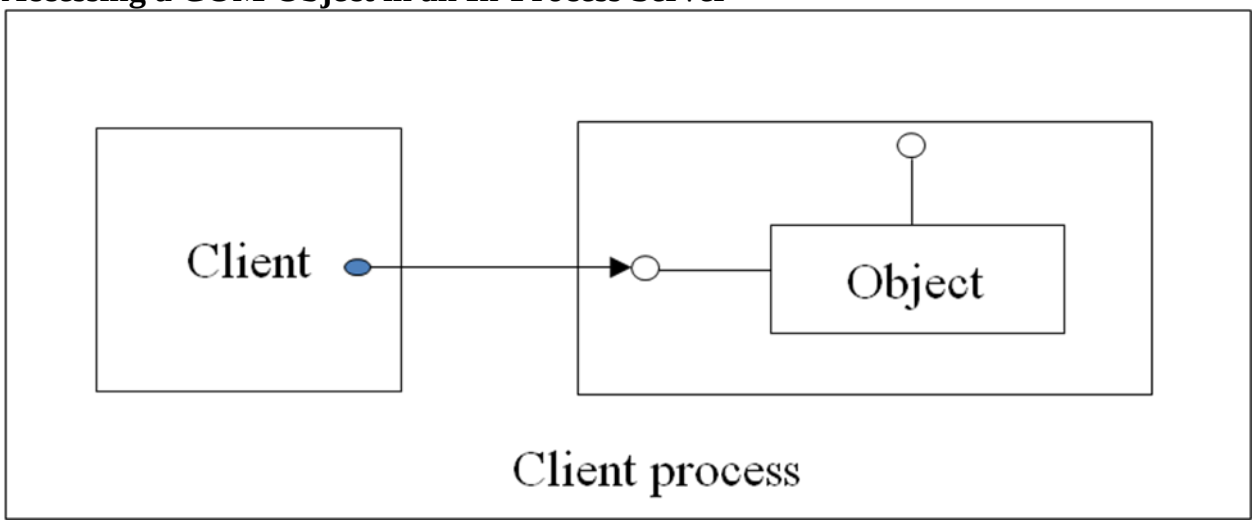
Marshaling and Type Information

- Marshaling makes that the client can invoke the methods in the same way, regardless of where the object is implemented

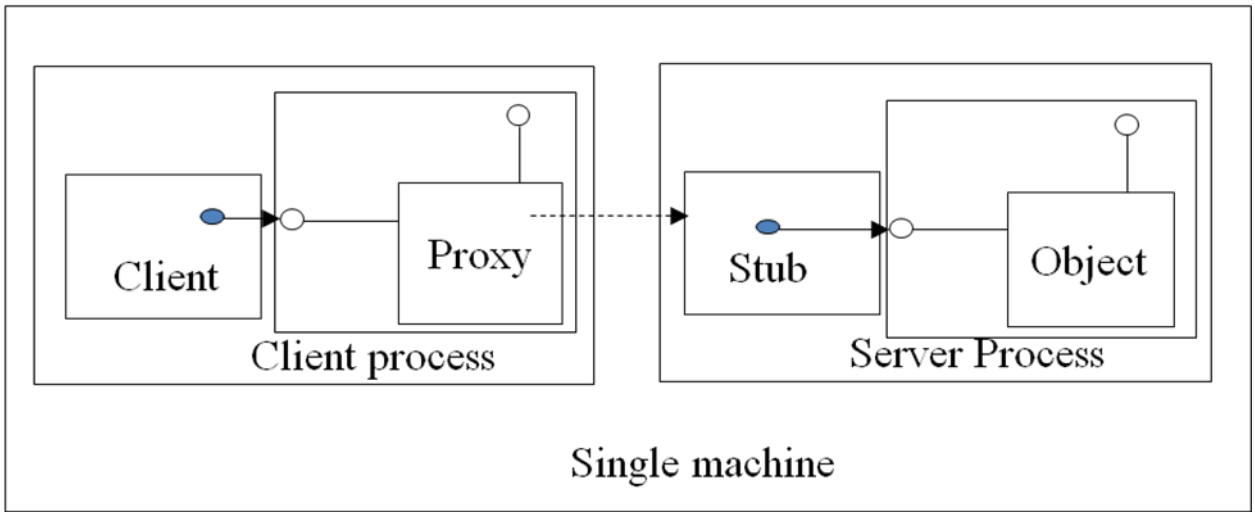
Kinds of COM Servers



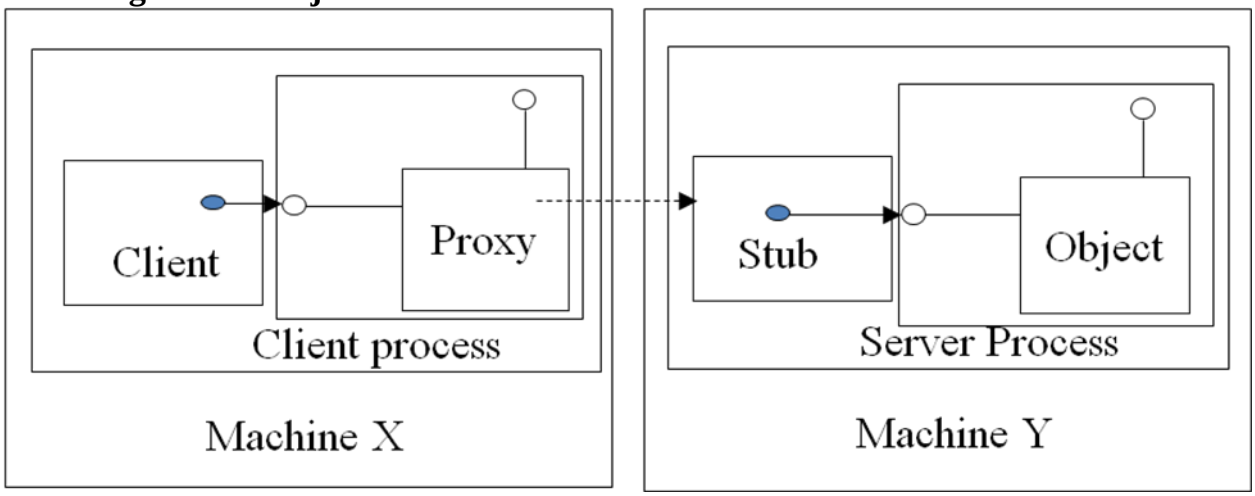
Accessing a COM Object in an In-Process Server



Accessing a COM Object in a Local Server

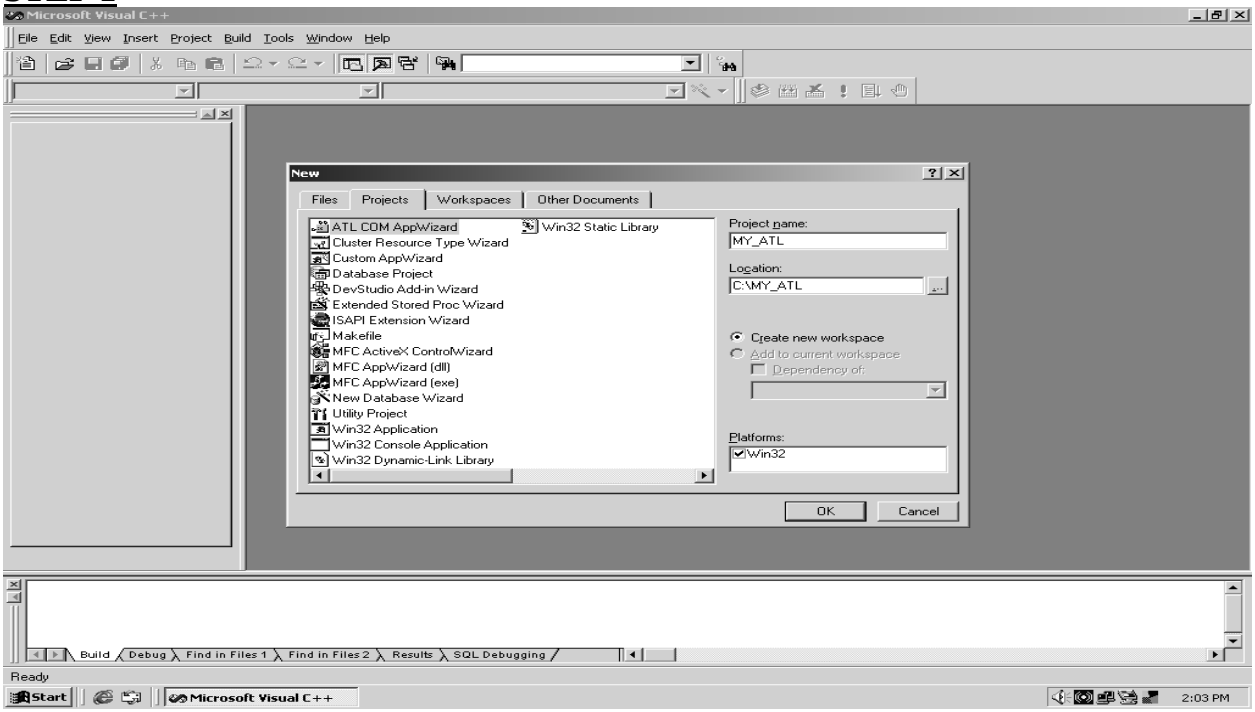


Accessing a COM object in a Remote Server

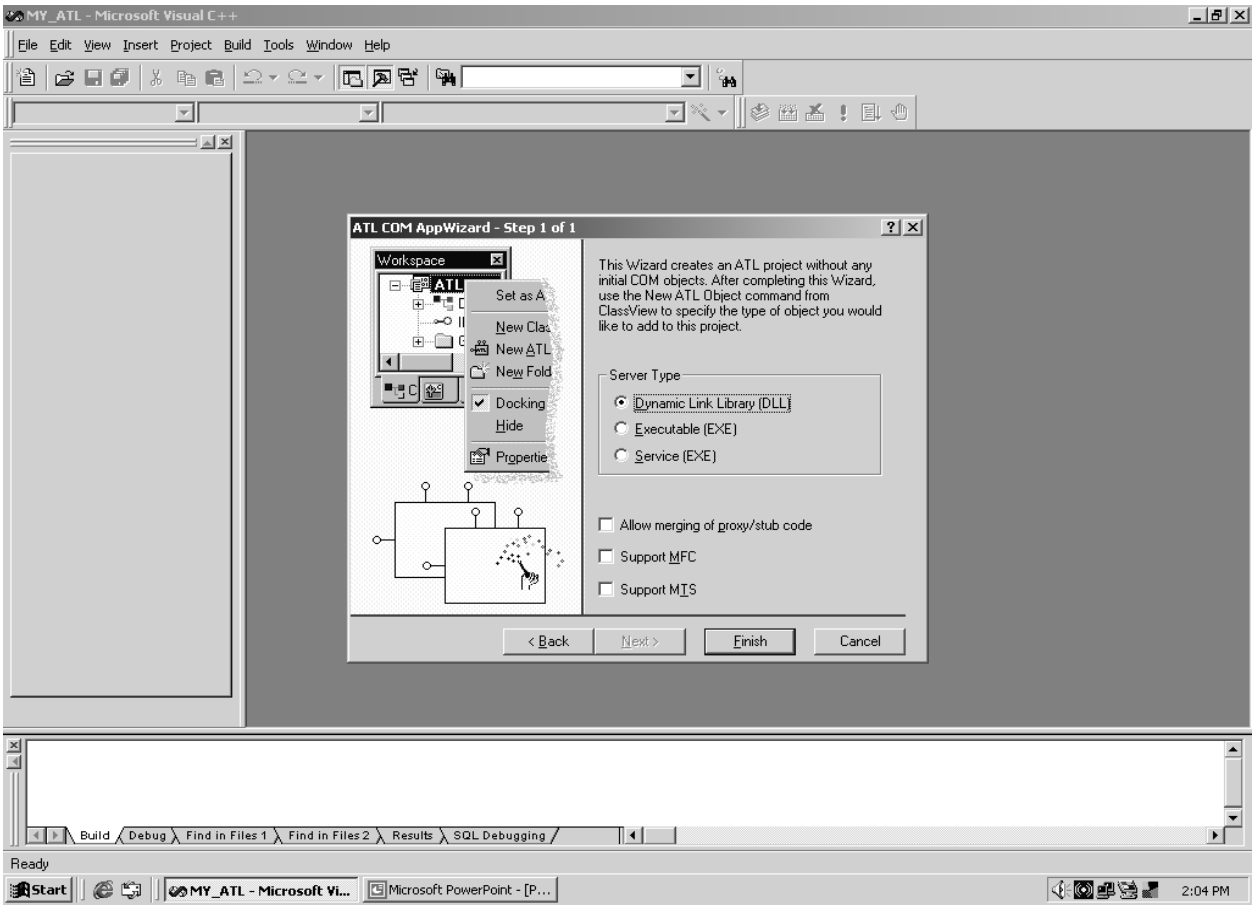


Steps to Create a COM using VC++

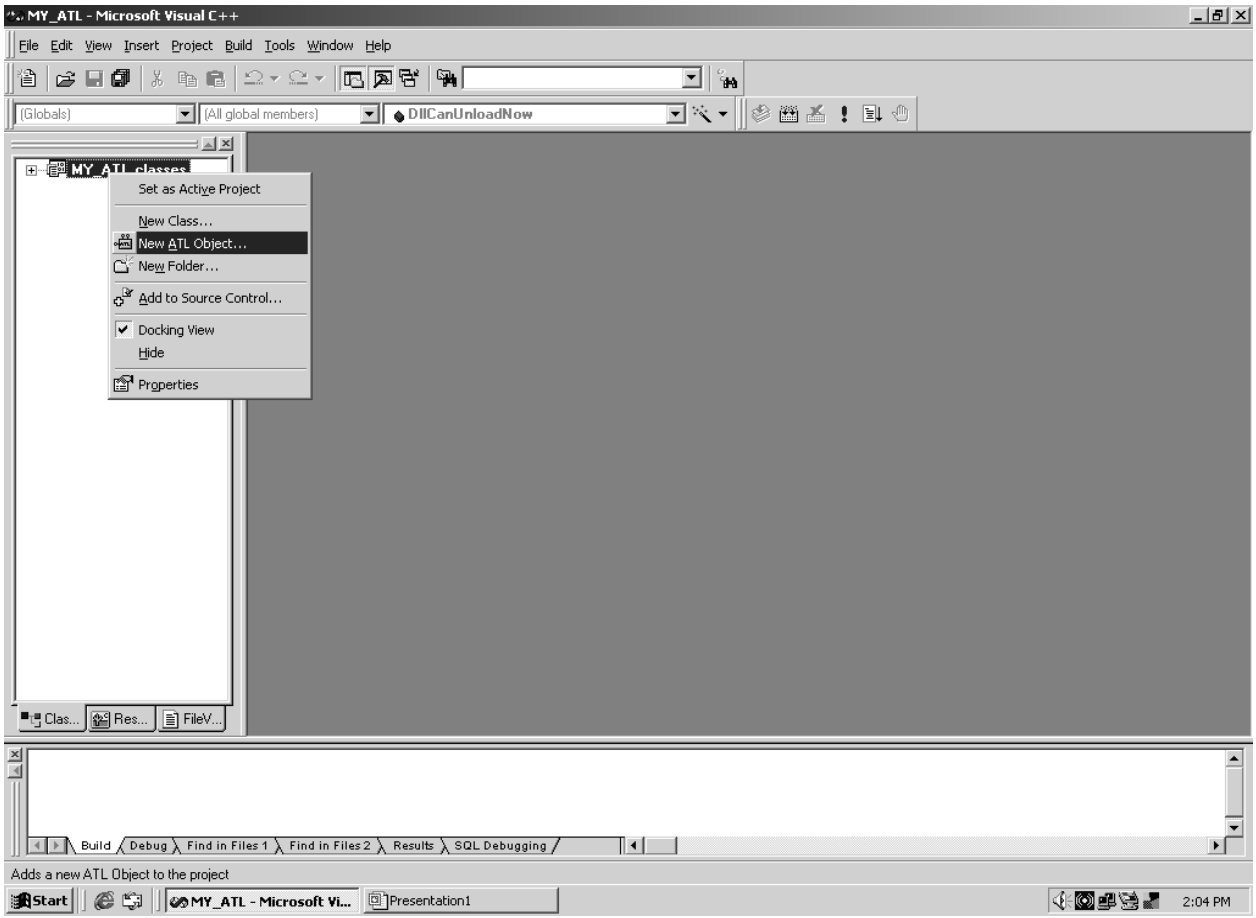
STEP 1



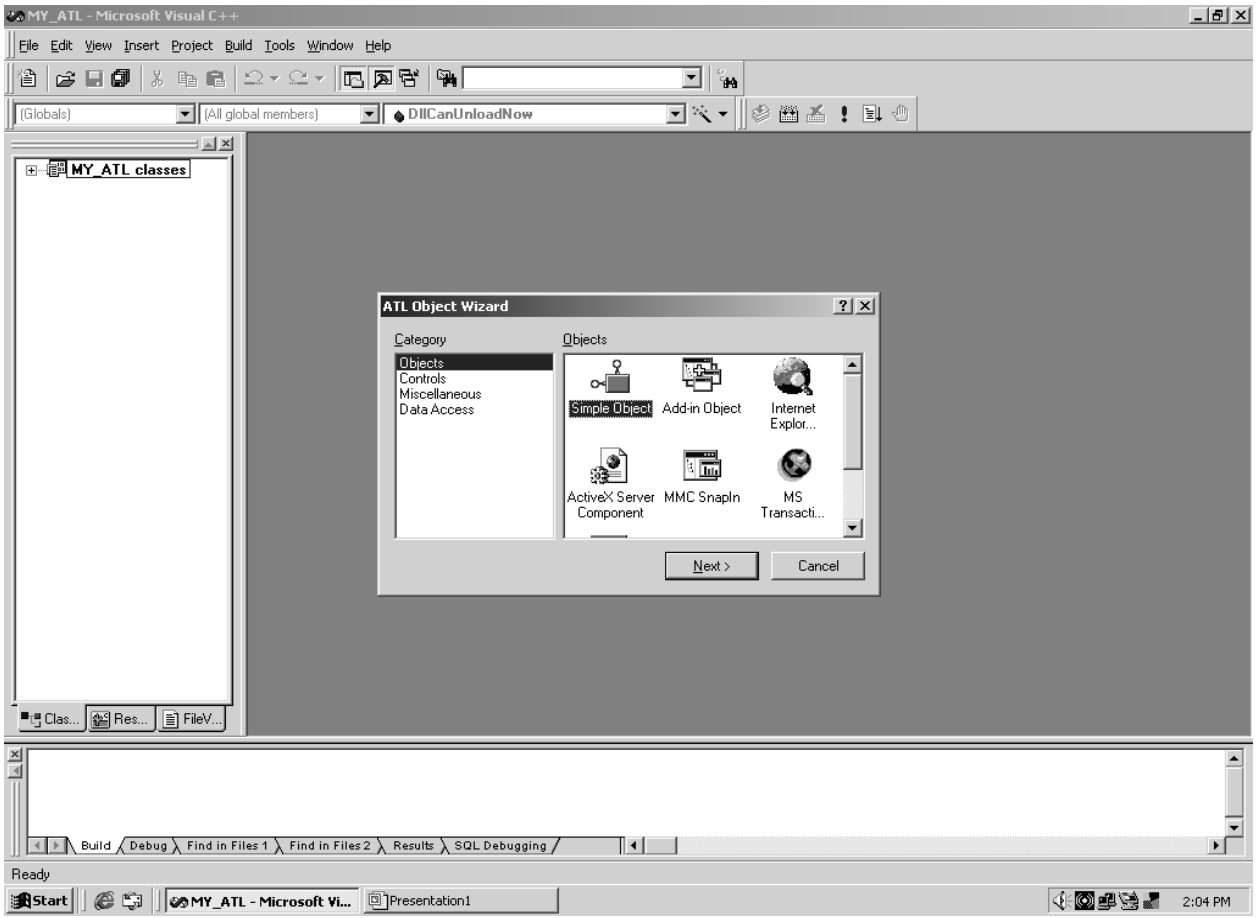
STEP 2


MY_ATL - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
ATL COM AppWizard - Step 1 of 1
Workspace
Set as A
New Clas
New ATL
New Fold
Docking
Hide
Propertie
This Wizard creates an ATL project without any initial COM objects. After completing this Wizard, use the New ATL Object command from ClassView to specify the type of object you would like to add to this project.
Server Type
Dynamic Link Library (DLL)
Executable (EXE)
Service (EXE)
Allow merging of proxy/stub code
Support MFC
Support MTS
< Back
Next >
Finish
Cancel
Build
Debug
Find in Files 1
Find in Files 2
Results
SQL Debugging
Ready
Start
MY_ATL - Microsoft Vi...
Microsoft PowerPoint - [P...
2:04 PM

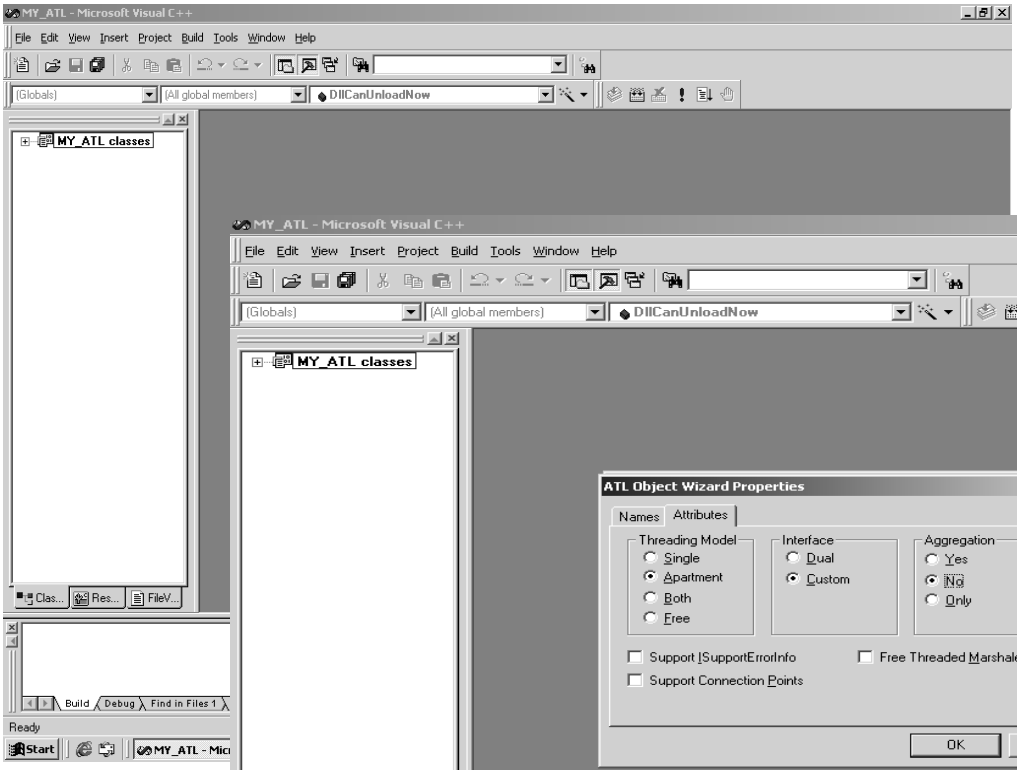
STEP 3


MY_ATL - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
(Globals)
(All global members)
DllCanUnloadNow
MY_ATL classes
Set as Active Project
New Class...
New ATL Object...
New Folder...
Add to Source Control...
Docking View
Hide
Properties
Clas...
Res...
FileV...
Build
Debug
Find in Files 1
Find in Files 2
Results
SQL Debugging
Adds a new ATL Object to the project
Start
MY_ATL - Microsoft Vi...
Presentation1
2:04 PM

STEP 4


MY_ATL - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
(Globals)
(All global members)
DllCanUnloadNow
MY_ATL classes
Clas...
Res...
FileV...
ATL Object Wizard
Category
Objects
Controls
Miscellaneous
Data Access
Objects
Simple Object
Add-in Object
Internet Explor...
ActiveX Server Component
MMC SnapIn
MS Transacti...
Next >
Cancel
Build
Debug
Find in Files 1
Find in Files 2
Results
SQL Debugging
Ready
Start
MY_ATL - Microsoft Vi...
Presentation1
2:04 PM

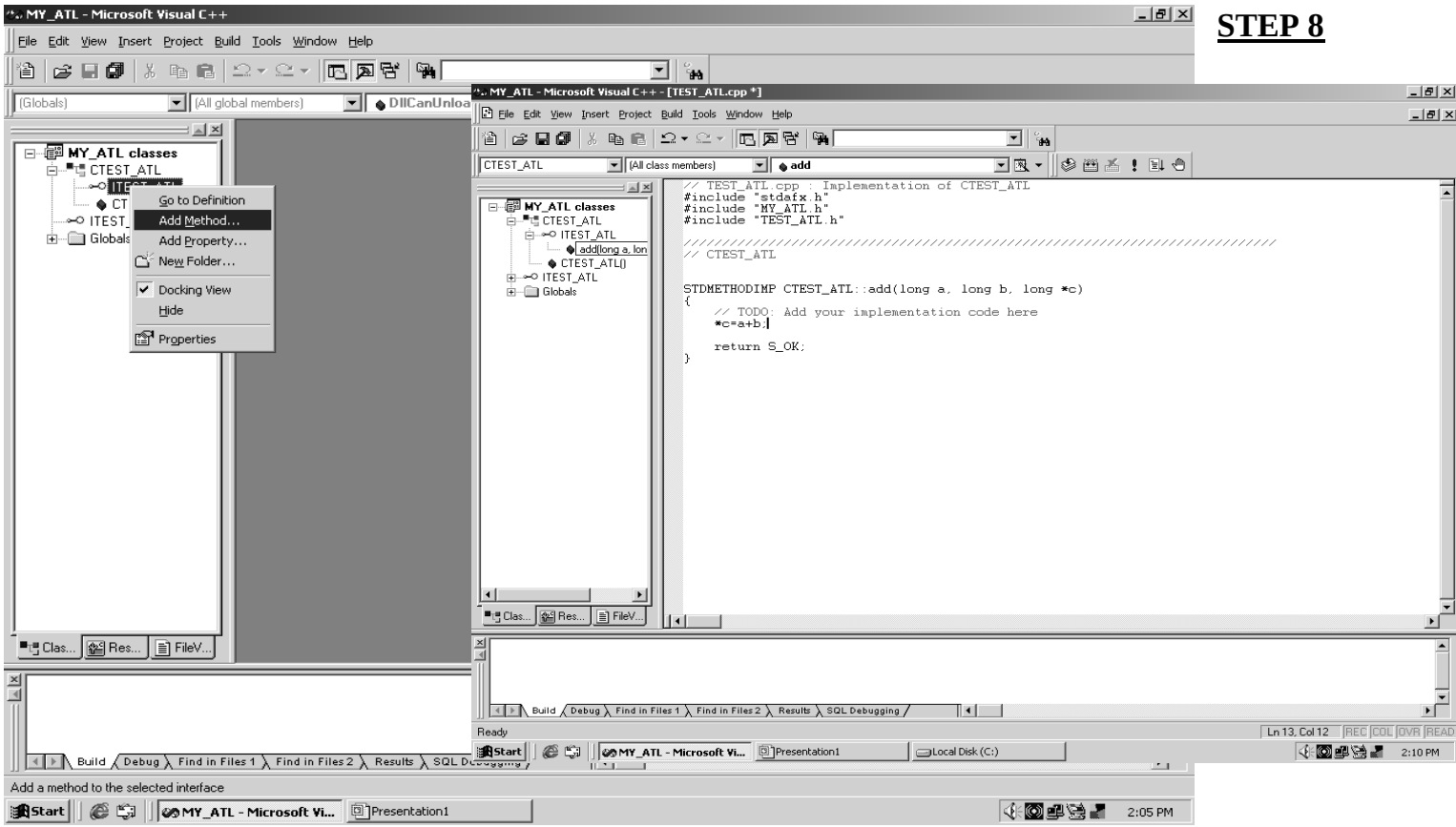
STEP 5



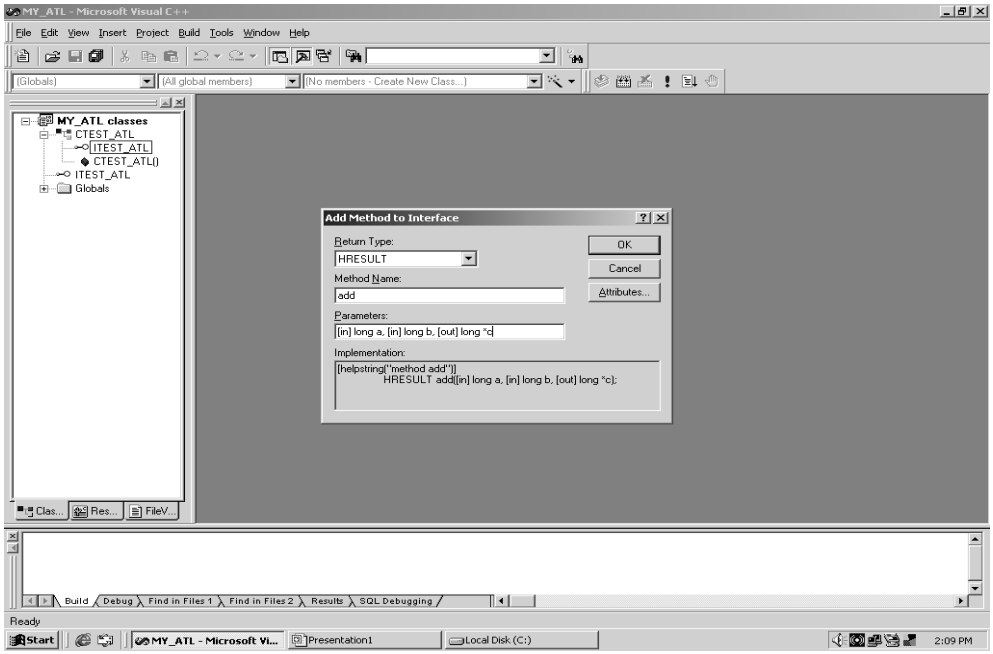
STEP 6

STEP 7

STEP 8



STEP 9



- SAVE
- COMPILE by F7

COM CREATED SUCESSFULLY

UNIT – IV

ACTIVEX AND OBJECT LINKING AND EMBEDDING

PART – A (2 MARKS)

1. Define ActiveX control
2. List out Calendar control's properties, methods and events.
3. Define Container
4. Define Event sink map
5. Define COM
6. Define Mini Server
7. Define Full Server
8. List out the Component States
9. What is the use of IUnknown interface?
10. What is class factory?
11. Define OLE
12. Define DCOM
13. What are main features of COM?

PART – B

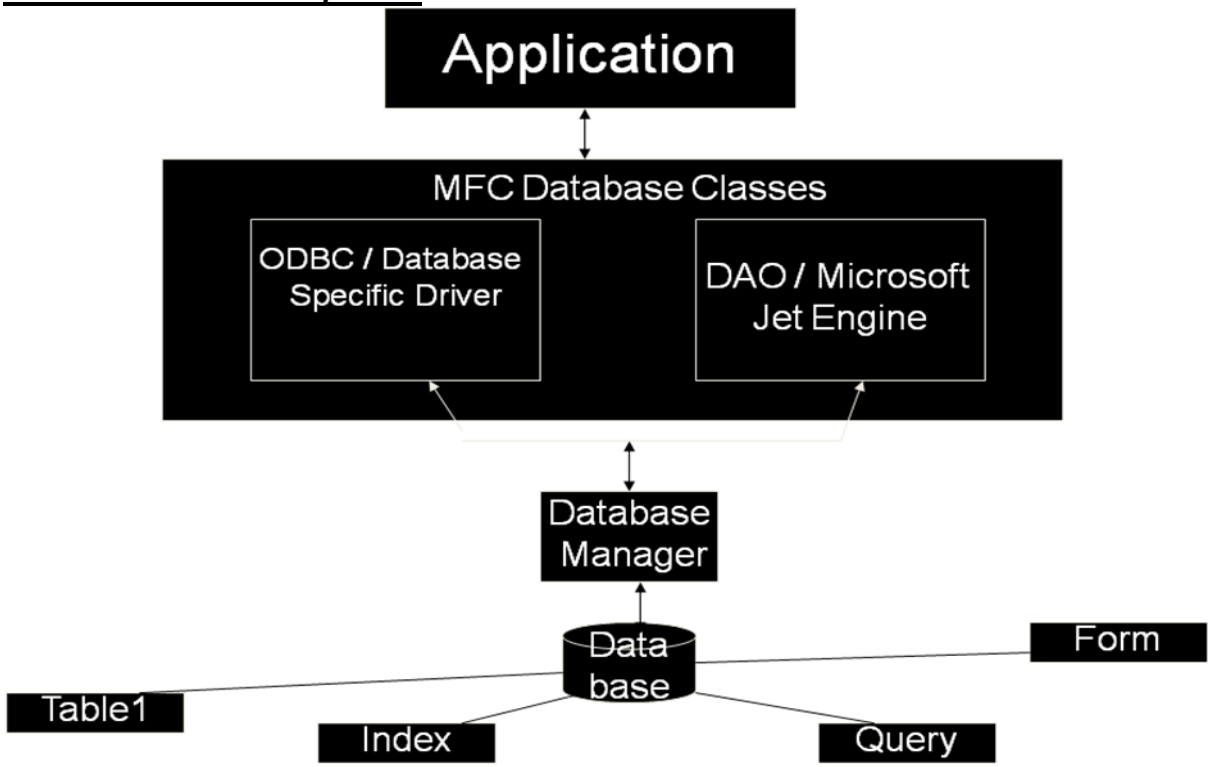
1. How the COM Client interacts with Inprocess Component. (16)
2. a. What are the steps involved to create an ActiveX control at run time (6)
b. What are the steps involved in OLE Drag & Drop (10)
3. Explain the features of OLE container – component interactions (16)
4. Explain in detail ActiveX control container programming with example (16)
5. Write a short notes on
a. IUnknown Interface and QueryInterface Member function (10)
b. Reference Counting (6)
6. Write a short notes on
a.Class Factory (8)
b.Containment & Aggregation Vs Inheritance (8)
7. a. Write a COM class using multiple inheritance approach (8)
b. Discuss the container interfaces (8)
8. a. Highlight the features of the control (8)
b. Explain the steps involved in the installation of ActiveX control(8)

UNIT V

Introduction

- Database is used to store data and
- Provide access to manipulate the data.
- **Use of standard file formats**
- Provides multiple user interaction
- ***Database with Visual C++***

Database Architecture of VC++



ODBC Architecture

- use the ODBC API to access data from a variety of different data sources
- Contains Driver Manager for performing the database activities.
- Supports various database drivers Microsoft SQL Server Oracle Microsoft Access Microsoft FoxPro
- Implemented by C native API

MFC classes for ODBC

- There are 3 different Built in classes provided by MFC
- CDatabase Manages a Connection to a data source. Work as a Database Manager
- CRecordSet Manages a set of rows returned from the database. CRecordView Simplifies the display of data from CRecordSet Object.

ODBC classes overview

- ***CRecordSet*** MFC Appwizard generates a CRecordSet derived class and return a pointer named m_pSet to our application program.
- **How to Map database values to Recordset Using Record Field Exchange** we can move the data back and forth from recordset to data base. The exchange is set up by implementing the CRecordset:: DoFieldExchange() function, and it maps the member variables of Recordset and Database.

```
void CChap21Set::DoFieldExchange(CFieldExchange* pFX)
{   //{{AFX_FIELD_MAP(CChap21Set)
    pFX->SetFieldType(CFieldExchange:: outputColumn);    RFX_Long(pFX, _T("[EmpId]"), m_EmpId);
    RFX_Text(pFX, _T("[EmpName]"), m_EmpName);
```

```
    RFX_Text(pFX, _T("[Salary]"), m_Salary);
//}}AFX_FIELD_MAP}
```

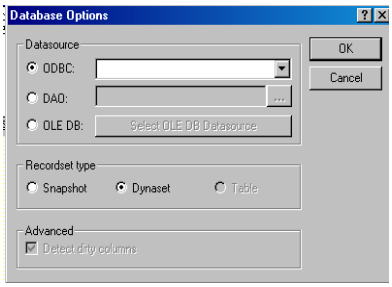
- CRecordset::GetFieldValue() Is a alternative for RecordFieldExchange Which enables you to retrieve the value of any field in a current View. Even if we not defined member variable OR set up of RFX. Using the column name or index to retrieve the data retrieve values as a CString or a CDBVariant object

CRecordset functions

- Provides various built in functions
- Table and ODBC related
- To Navigating data in the recordset
- To Manipulating the data's in record set
- Bookmark the records in a recordset
- Tabl and ODBC related Functions CRecordSet:: GetSQL() Returns the Entire SQL String. GetTableName() Returns the table name used. GetODBCFieldCount() Returns the total no of columns returned by the query Close() Close the database connection Open() Reconnect / connect the data base to the program
- Navigating the data in recordset MoveNext() MovePrev() MoveLast() MoveFirst() CanScroll() Check the recordset having only forward only cursor SetAbsolutePosition() Move to specific rows in Record set Which takes Zero based index into the record set.
- To Manipulating the data's in record set Delete() Delete the current row & set the member variables NULL. AddNew() Create a new row with field values are empty. CanAppend() Used to check whether record set provides adding of records Edit()To edit or modify the current record details Update() Used to update the record set when a new record is added / existing record is edited.

Recordset selection

- Visual C++ provides 3 types of Recordset They are differ in speed versus features Snapshot Dynaset Table



- Snapshot Download the entire query in one shot Have data as a static copy When any changes made to the database will not reflected to the current Application. Occupy more memory to hold the data.
- Dynaset Only the records you actually need to fill the screen will get downloaded. Take less time to reflect. Constantly resynchronizes the recordset, so that any changes will reflected immediately.
- The snapshot and Dynaset work at the record level. ODBC will only support this two options.
- Table Work with table level and supported by DAO. Places the contents of the query into Temporary table. Have a problem with updation.

CRecordView

- is basically a form view

- make it easier to display data from a recordset
- enables you to use dialog data exchange to display data directly in a dialog box from the recordset
- Functions of CRecordView class
- DoDataExchange() Perform dialog data exchange. In a Normal version move data between control and member variable. It will move data between the view controls and column data member variables of CRecordset. Sample code:

```
  void CChap21View::DoDataExchange(CDataExchange* pDX)
 {
  CRecordView::DoDataExchange(pDX);    //{{AFX_DATA_MAP(CChap21View)
     DDX_FieldText(pDX, IDC_DEPTCODE, m_pSet->m_Dept, m_pSet);    DDV_MaxChars(pDX, m_pSet-
>m_Dept, 10);    DDV_MaxChars(pDX, m_pSet->m_EmpName, 50);
//}}AFX_DATA_MAP
```

- OnGetRecordSet() Retrieve a pointer of the CRecordset. The default implementation supplied by Class Wizard returns the pointer stored in CRecordView ::m_pSet OnMove() takes only one parameter, specifying where to move. This can be one of the following constants: ID_RECORD_FIRST ID_RECORD_LAST ID_RECORD_NEXT ID_RECORD_PREV

CDatabase

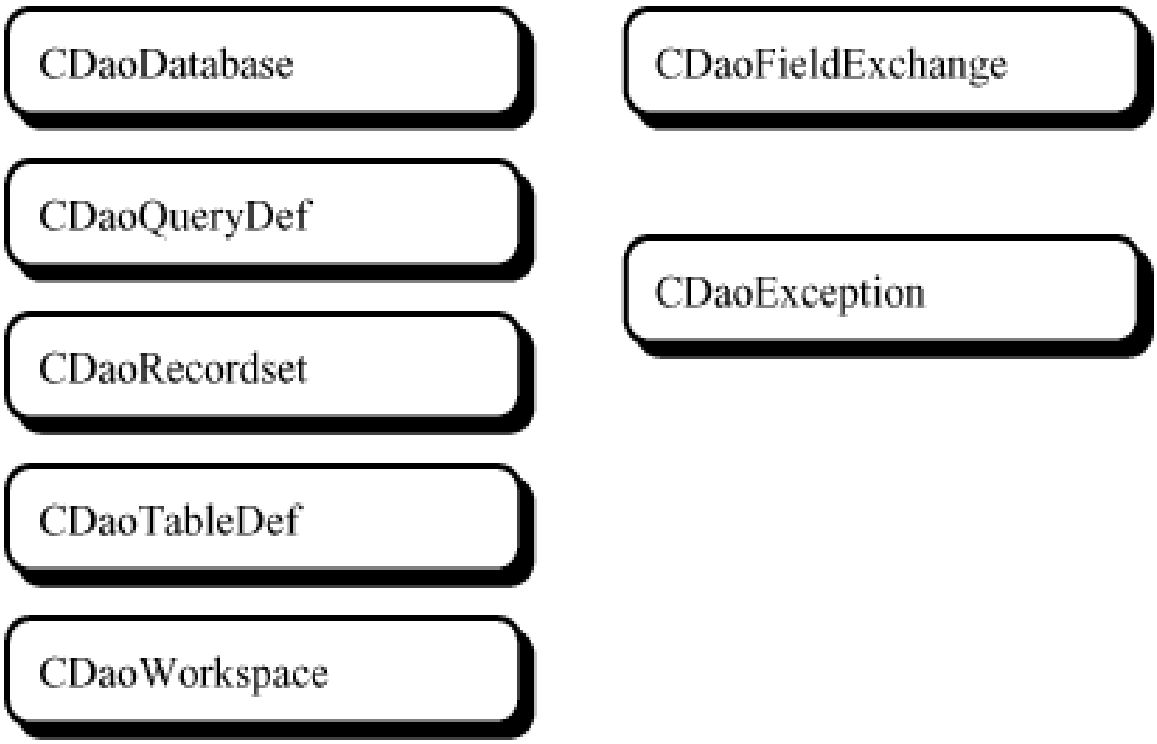
- is used to encapsulate your application's dealings with a connection to the database
- Perform ODBC C API connection Handles.
- We can retrieve CDatabase object associated with CRecordset by m_pSet->m_pDatabase variable in CRecordset
- Used to execute SQL statements void ExecuteSQL(LPCSTR sqlstmt) Takes SQL String & execute it against the current datasource Does not return error, if any run time error occurs, CDBException will be thrown
- Transaction with CDatabase Enables to execute a series of SQL statements as a single operation. One of the operation fails, rest of all can be undone. This is most useful future for doing related updation to various tables at the same time. CanTransact() BeginTrans() Tells transaction process starts. ExecuteSQL() CommitTrans() Rollback() The functions work properly, depends on the ODBC driver support.
- This example shows a simple transaction involving a row insertion made by calling ExecuteSQL():

```
    try {
     m_pSet->m_pDatabase->BeginTrans();
     m_pSet->m_pDatabase->ExecuteSQL( "INSERT INTO Employee VALUES ('Joe Beancounter', 'Accounting',
80000)");
     m_pSet->m_pDatabase->CommitTrans();
          }
    catch(CDBException *pEx)
    {
      pEx->ReportError();
     m_pSet->m_pDatabase->Rollback();
     }
```

DAO

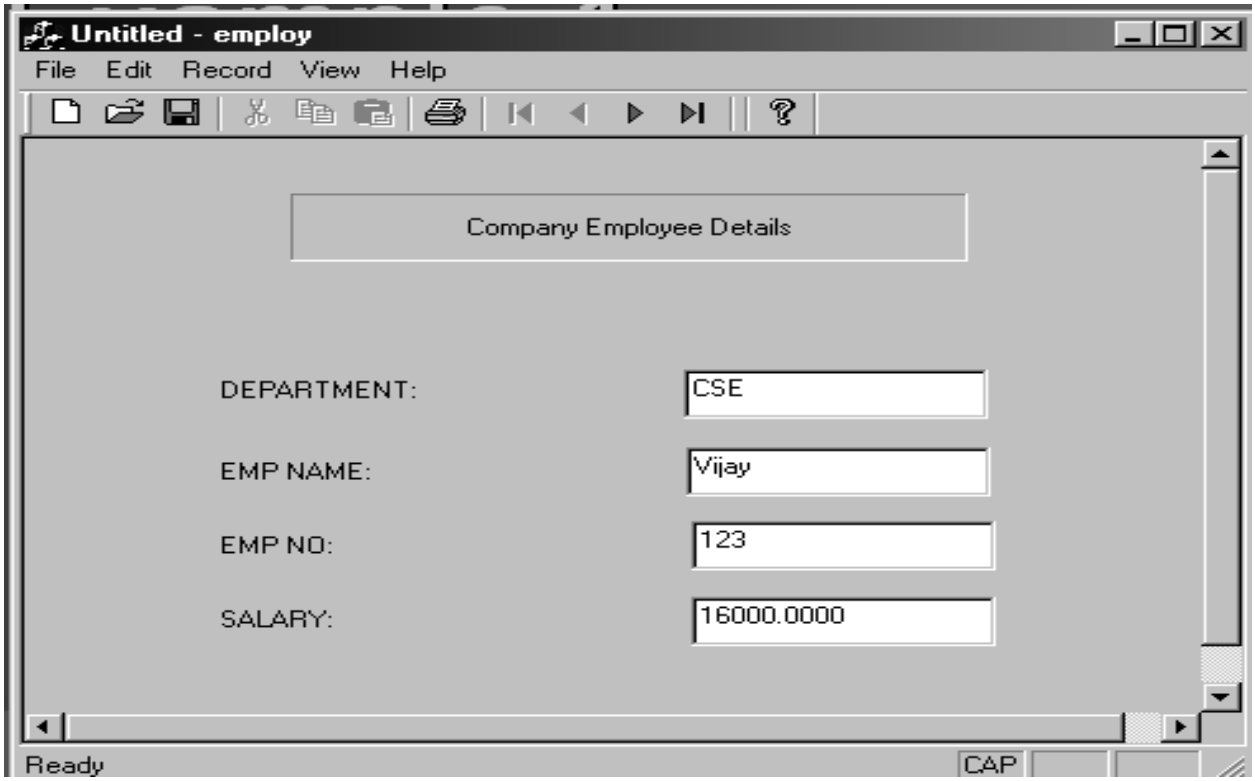
- Data Access Object
- is supplied in the form of redistributable components
- enable you to access and manipulate databases through the Microsoft Jet database engine.
- Similar to ODBC

- Won't support Remote Communication
- Is based on OLE.
- ***DAO Classes***

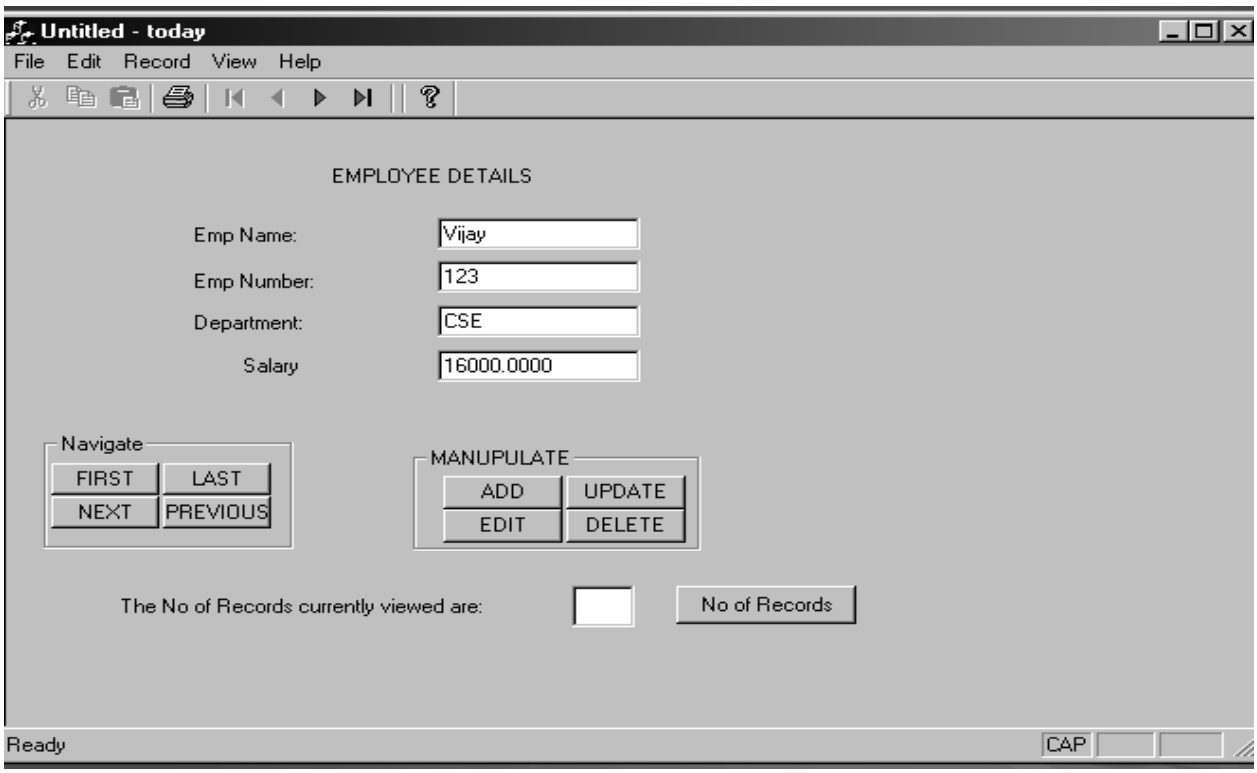


- ***CDaoRecordset*** **Just like CRecordset Object in ODBC The navigation functions include Find, FindFirst, FindLast, FindNext, and FindPrev;and Move, MoveFirst, MoveLast, MoveNext, and MovePrev. Data update functions include AddNew,CancelUpdate, Delete, Edit, and Update.**
- ***CDaoDatabase***
- **represents a connection to a database** A connection is created by calling CDaoDatabase::Open and terminated by calling CDaoDatabase::Close. A new database can be created by calling CDaoDatabase::Create. DeleteTableDef () **Deletes a DAO TableDef object and also the underlying table and all its data from the database.**
- ***CDaoWorkspace***
- **represents database sessions** created by calling CDaoWorkspace::Create An existing workspace object can be opened by calling CDaoWorkspace::Open
- ***CDaoQueryDef*** represents query definitions To create a new query definition CQueryDef::Create to access a already existing query definition CQueryDef::Open to execute an action query that modifies the data in the database CQueryDef::Execute
- ***CDaoTableDef*** represents table definitions open an existing table definition in a database by calling CDaoTableDef::Open A new table definition can be created by calling CDaoTableDef::Create Fields can be created and deleted by calling CreateField and DeleteField member functions
- CDaoFieldExchange is used in calls to DaoRecordset::DoFieldExchange
- ODBC Vs DAO
- when you only need access to data in a format that the Microsoft Jet engine can read directly (Access format, Excel format, and so on) the obvious choice is to use the DAO Database Classes.

- More complex cases arise when your data exists on a server or on a variety of different servers . ODBC provides facilities to access and perform complex join operations.
- Example 1
- Create a Visual C++ application using Appwizard to connect the Access database and display the records.

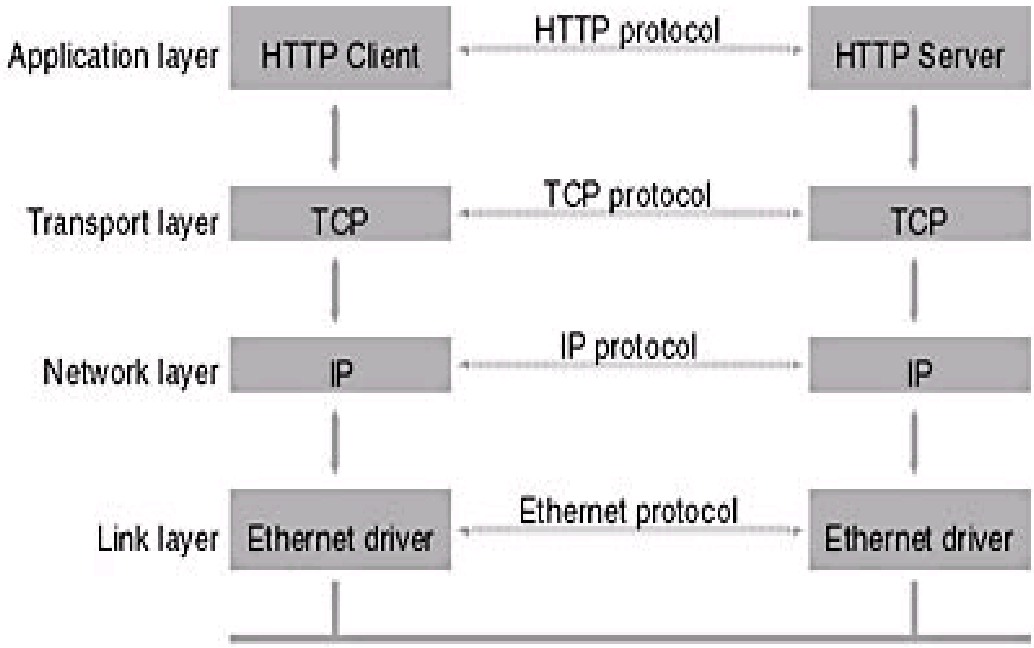


- Example 2
- Create a Visual C++ application using Appwizard to connect the Access database and perform Navigation and Manipulation operation using Dialog Controls.

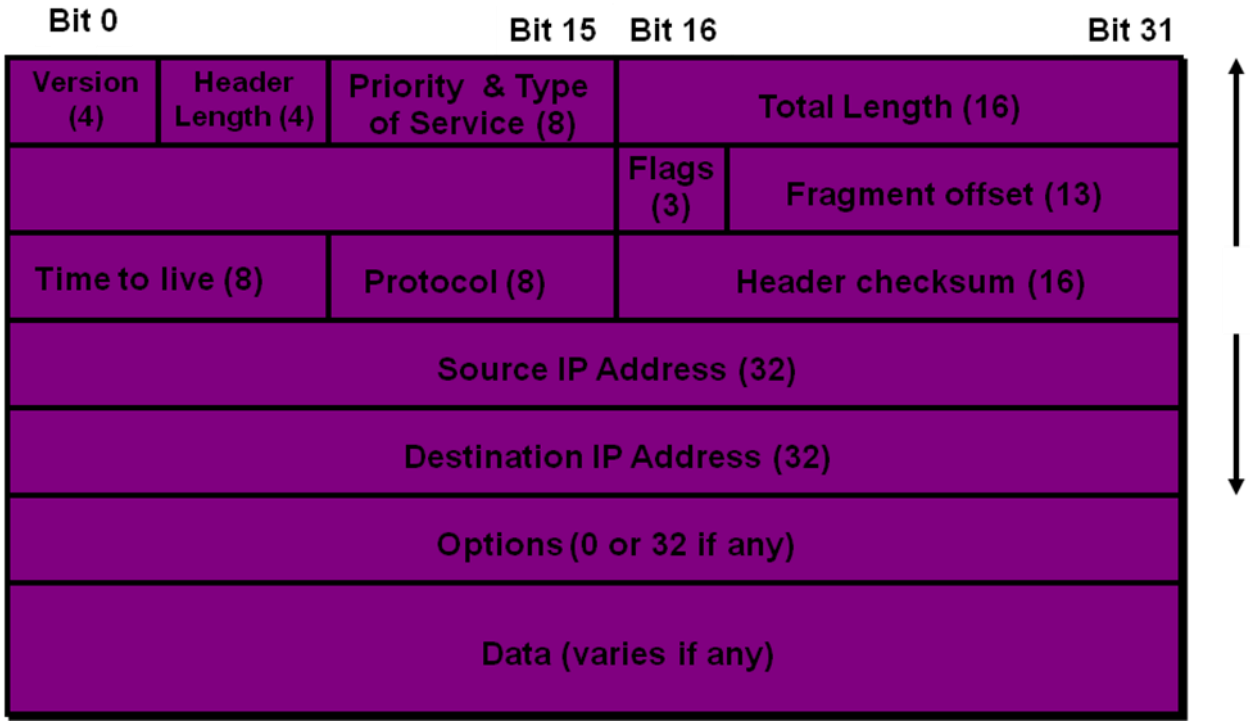


Network Issues

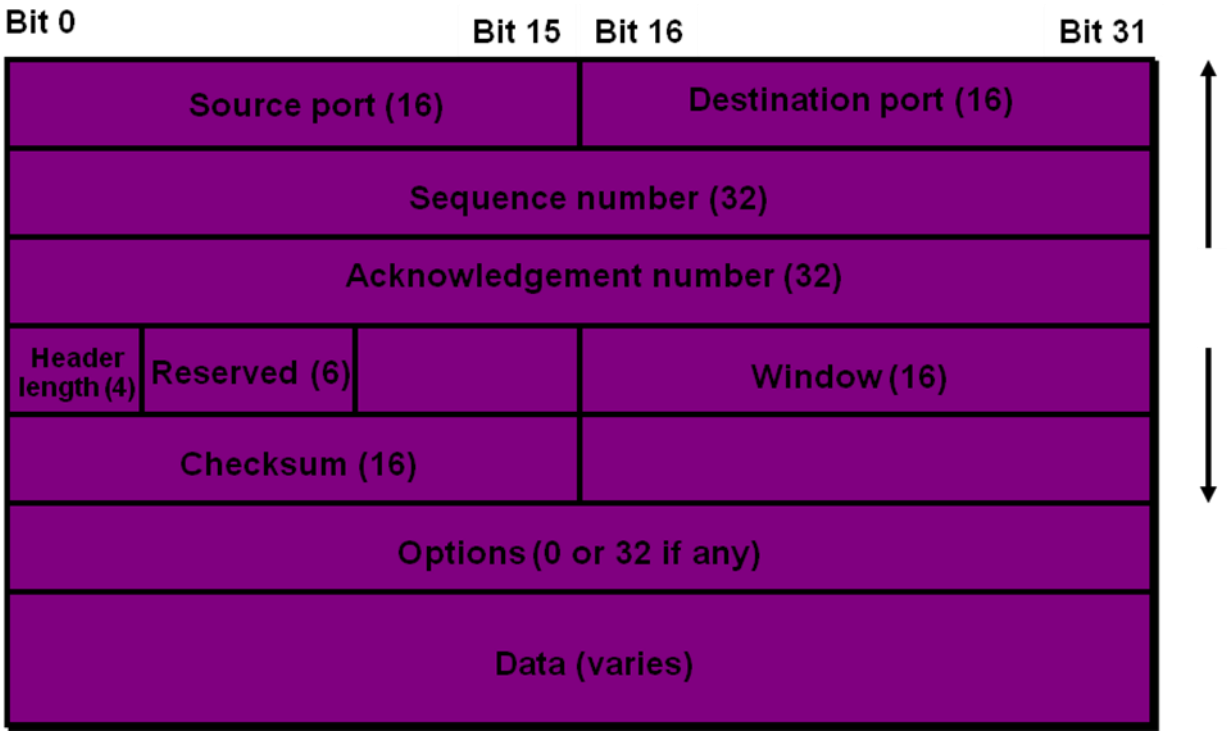
- **Network protocols – Layering**



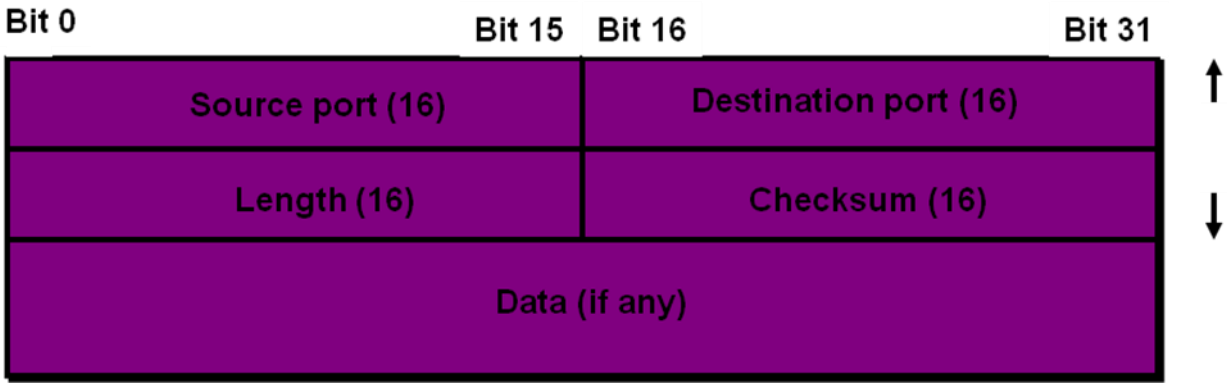
Internet Protocol



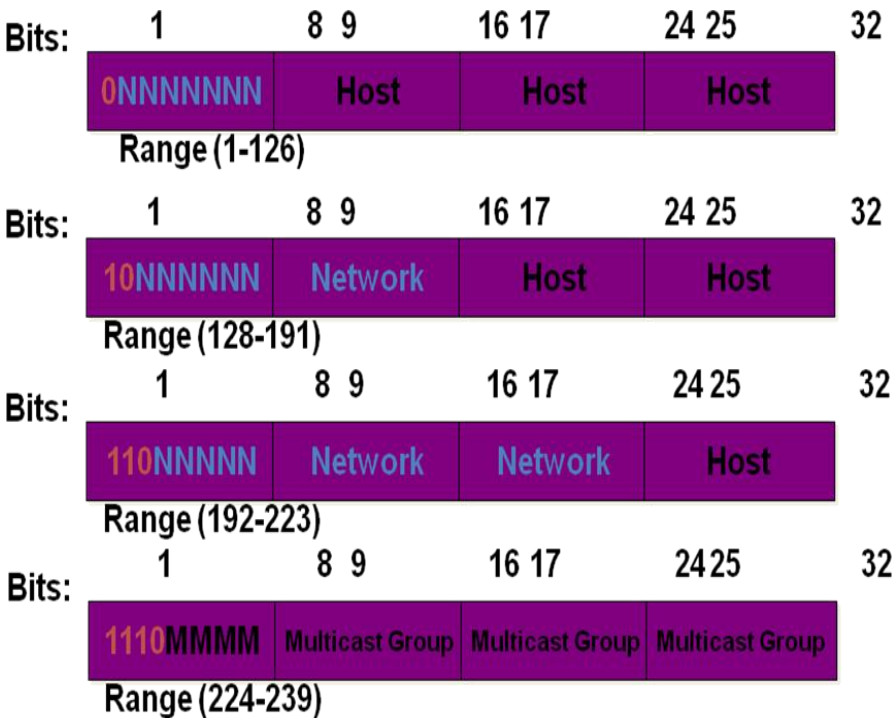
Connection-oriented protocol – TCP



Connectionless –UDP



IP Address



Network Byte Order

- All values stored in a sockaddr_in must be in network byte order.
  - sin_port a TCP/IP port number.
  - sin_addr an IP address.

Network Byte Order functions

- '**h**' : host byte order '**n**' : network byte order
- '**s**' : short (16bit) '**l**' : long (32bit)
- **uint16_t htons(uint16_t);**
- **uint16_t ntohs(uint_16_t);**
- **uint32_t htonl(uint32_t);**
- **uint32_t ntohl(uint32_t);**
- File System –NTFS Vs FAT

NTFS	FAT
More secured	Less Secured
User permission for individual files and folders	No individual user permissions
Used with Win 95,98	Used with Win NT and above

Socket

- A socket is an abstract representation of a communication endpoint.
- Sockets work with Unix I/O services just like files, pipes & FIFOs.
- Sockets have special needs:
  - ☐ establishing a connection
  - ☐ specifying communication endpoint addresses

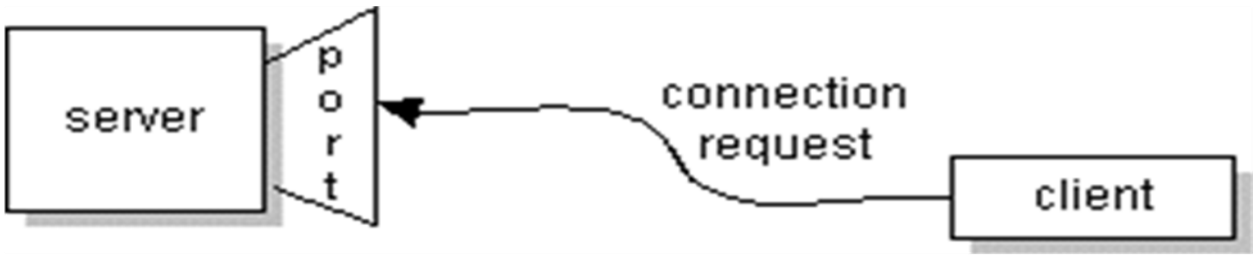


Figure A

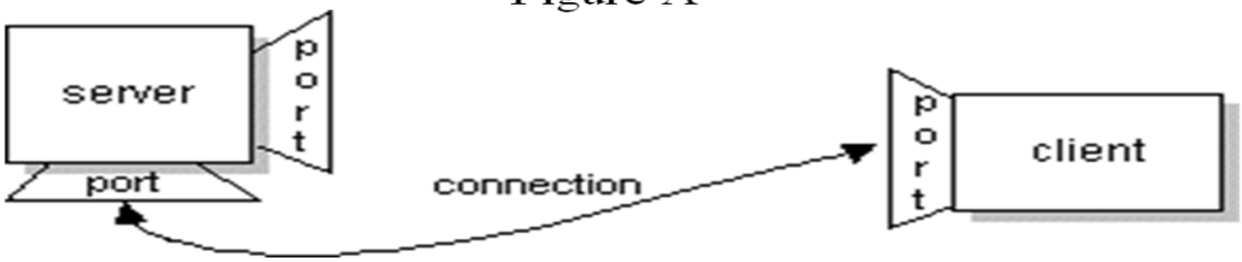
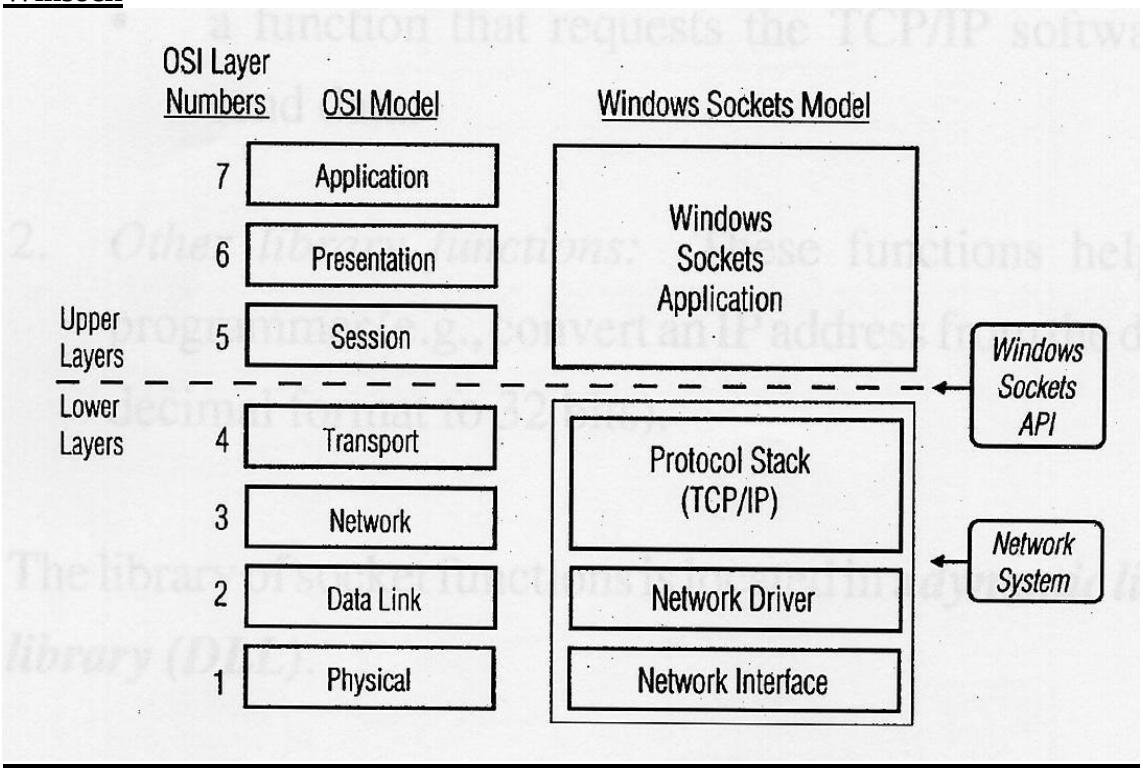


Figure B

Winsock



3 Types of Socket

- Stream sockets interface to the TCP (transmission control protocol).
- Datagram sockets interface to the UDP (user datagram protocol).
- Raw sockets interface to the IP (Internet protocol).

MFC Winsock classes

- CAsyncSocket - CAsyncSocket is a thin wrapper around the C API
- CSocket – base class
- CBlockingSocket - A thin wrapper around the Windows API.

Feature :Exception throwing and time outs

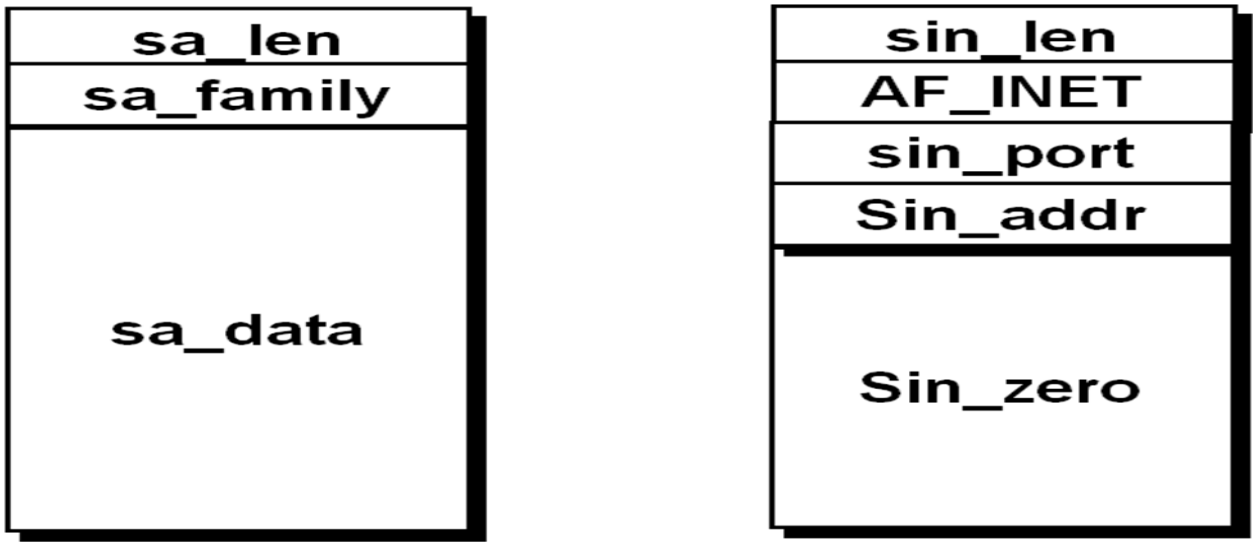
- CHttpBlockingSocket –read http data
- Helper classes:CSockAddr & CBlockingSocketException
- CSockAddr

CSockAddr

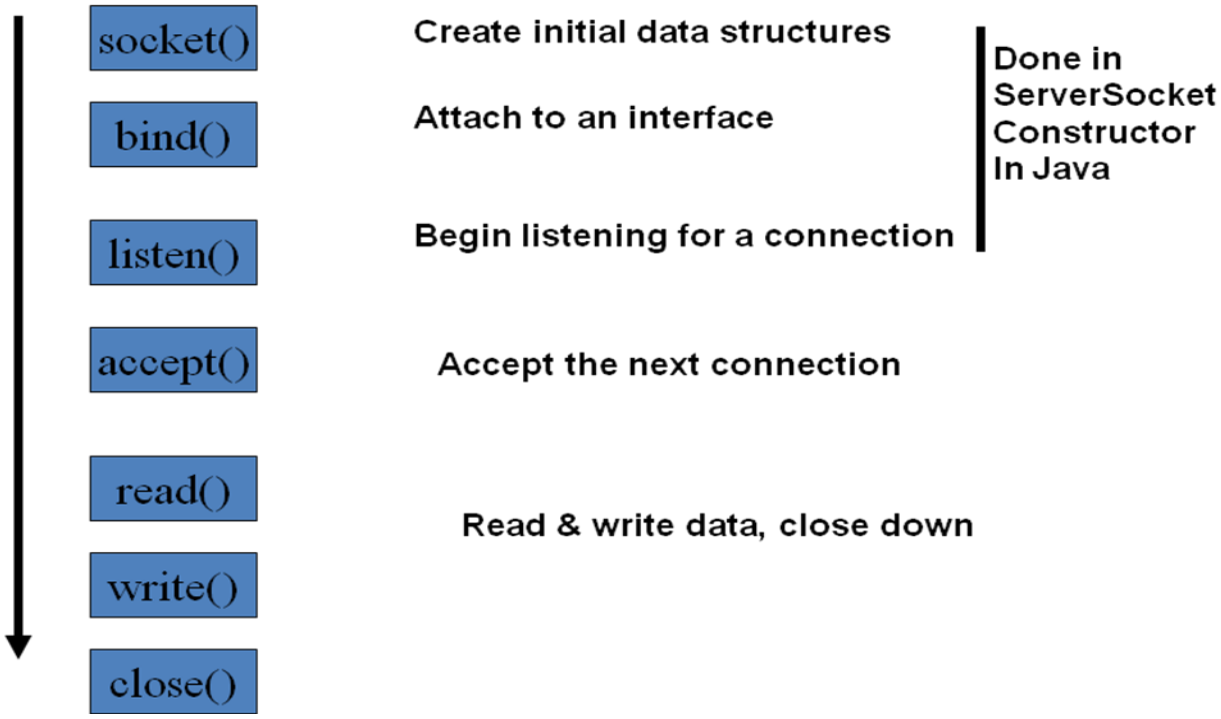
```
struct sockaddr_in {
        uint8_t         sin_len;
        sa_family_t     sin_family;
        in_port_t       sin_port;
        struct in_addr  sin_addr;
        char            sin_zero[8];
};
```

A special kind of sockaddr structure

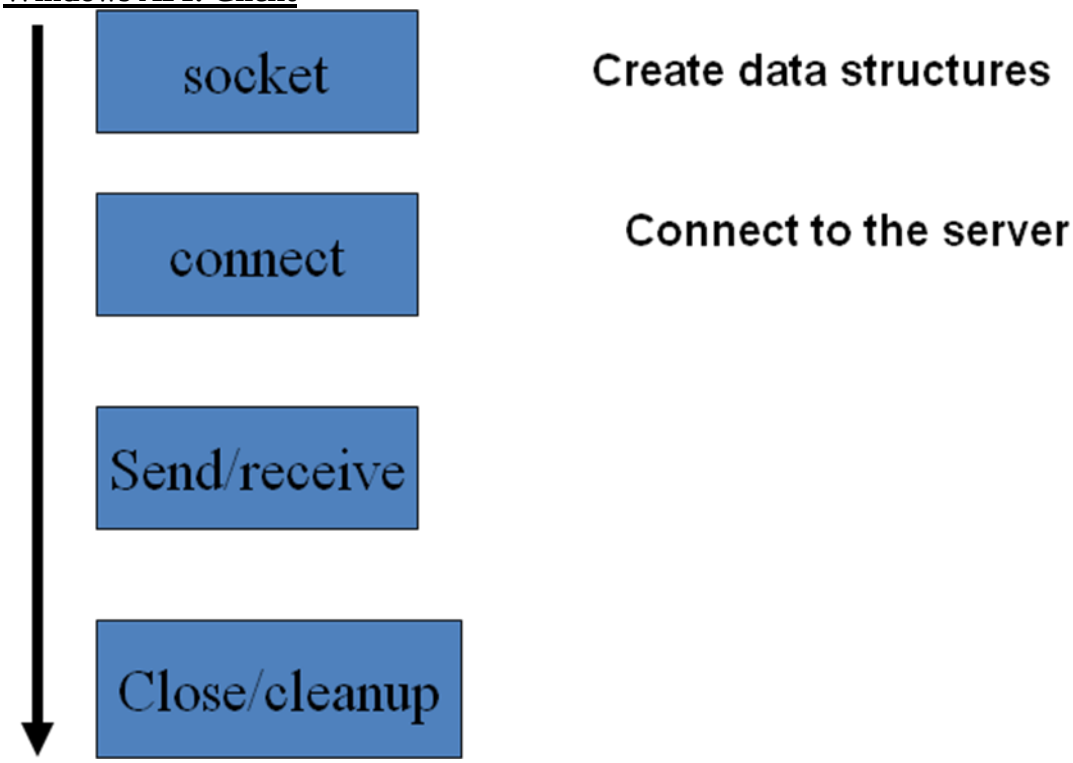
sockaddr ***Sockaddr_in***



Windows API: Server



Windows API: Client



Class definition

- class CSockAddr : public sockaddr_in {

```
public:
// constructors
 CSockAddr()
  {
sin_family = AF_INET;
sin_port = 0;
sin_addr.s_addr = 0;     } // Default
CSockAddr(const SOCKADDR& sa) { memcpy(this, &sa,       sizeof(SOCKADDR)); }
CSockAddr(const SOCKADDR_IN& sin) { memcpy(this, &sin,       sizeof(SOCKADDR_IN)); }
CSockAddr(const ULONG ulAddr, const USHORT ushPort = 0)
 // parms are host byte ordered
 {
 sin_family = AF_INET;
 sin_port = htons(ushPort);
  sin_addr.s_addr = htonl(ulAddr);
 }
 CSockAddr(const char* pchIP, const USHORT ushPort = 0)
 // dotted IP addr string
 {
sin_family = AF_INET;
sin_port = htons(ushPort);
 sin_addr.s_addr = inet_addr(pchIP);
  } // already network byte ordered
```

WinInet

- WinInet is a higher-level API ,but it works only for HTTP, FTP, and gopher client programs
- Benefits

  Caching
  Security
  Web proxy access
  User friendly

MFC WinInet Classes

- ***CInternetSession***
- ***CHttpConnection***
- ***CFtpConnection***
- ***CGopherConnection***

Moniker

- A moniker is a COM object that holds the name (URL) of the object, which could be an embedded component but more often is just an Internet file.
- Monikers implement the *IMoniker* interface, which has two important member functions: *BindToObject - object into running state*

BindToStorage - object data can be read

Internet Information Server

IIS

3 servers
High performance Internet/Intranet server
Special kind of Windows program- service

Allows to define virtual web server
Provides for Strong Authentication
Allows IP source filtering
Scaled down –Personal Web Server

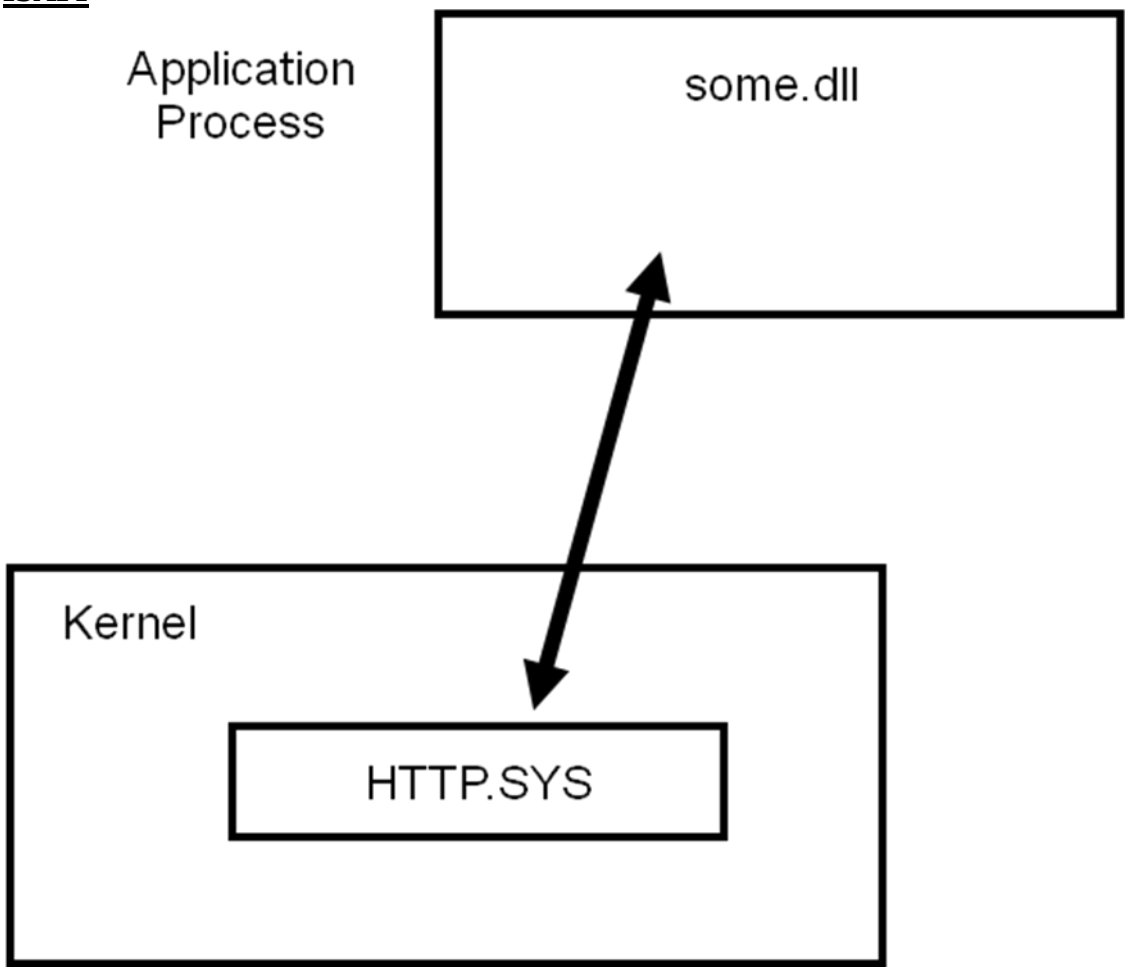
ISAPI

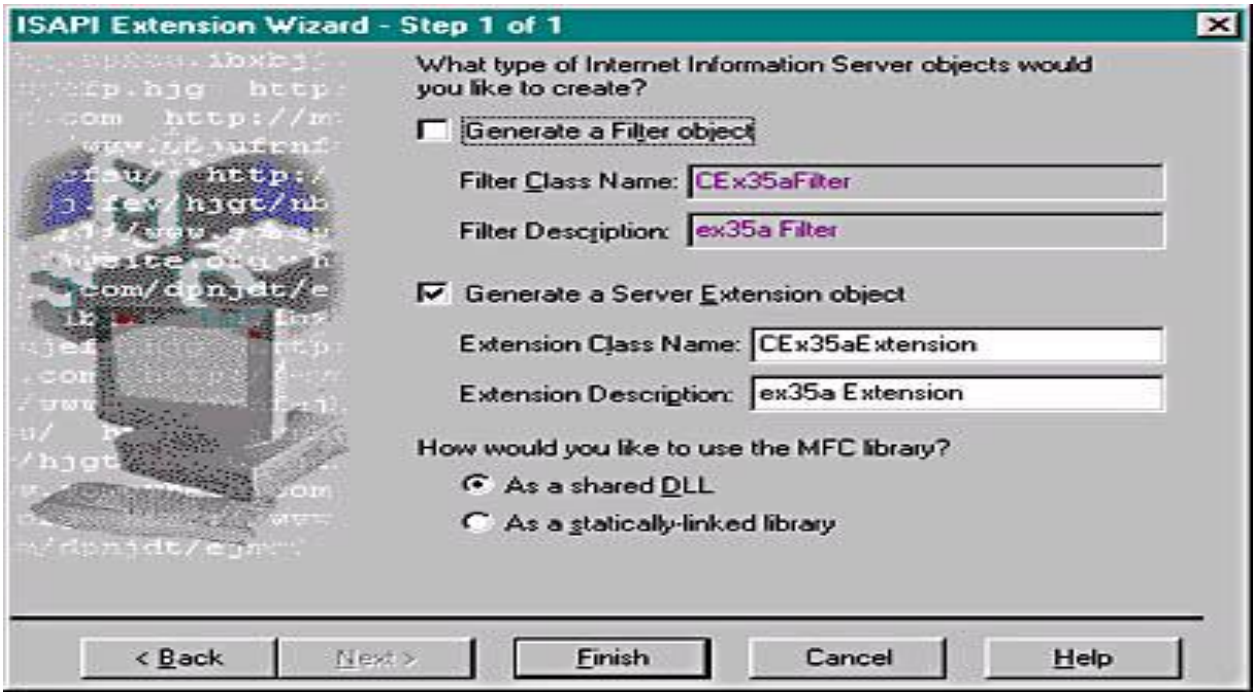
An ISAPI server extension can perform Internet business transactions such as order entry. It is a program runs in response to a GET or POST request from a client program
An ISAPI filter intercepts data traveling to and from the server and thus can perform specialized logging and other tasks
ISAPI server extension and ISAPI filter are DLLs.
ISAPI DLLs are usually stored in a separate virtual directory on the server.
These DLLs must have execute permission but do not need read permission.

HTTP.SYS

Called by TCP/IP when data arrives on a port associated with IIS
Reads HTTP headers into Kernel memory
Maps port+hostname+application to a running process
Passes request to that process

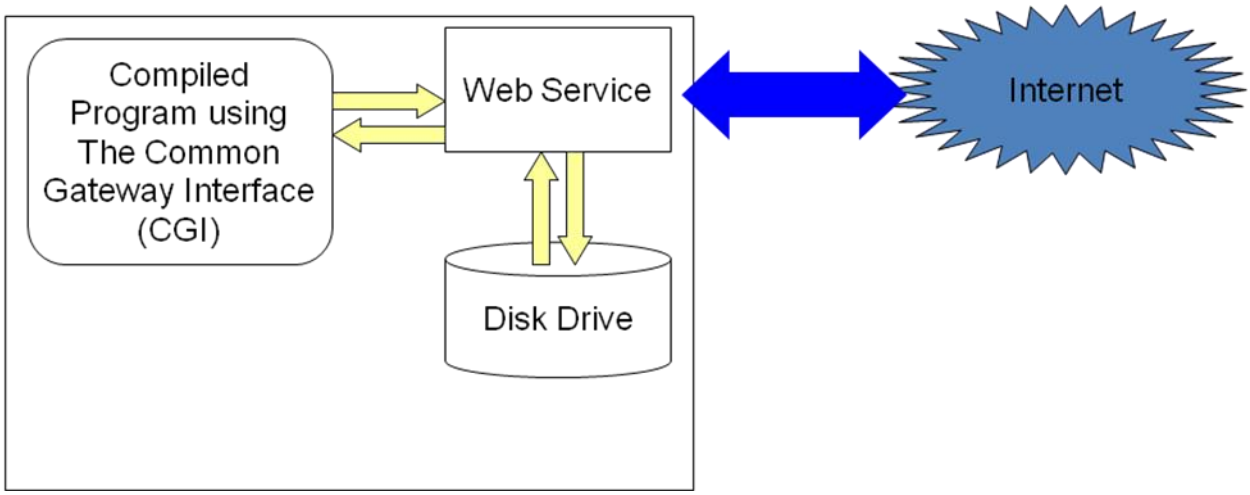
ISAPI





Web services

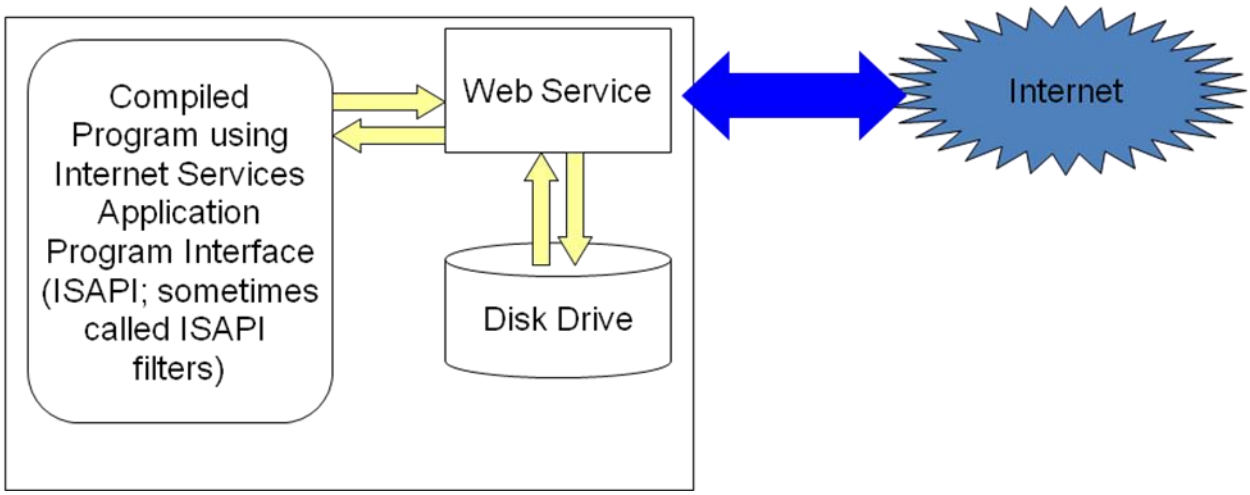
- CGI programs allow dynamic webpage content; HTML is built when a page is requested, instead of existing statically on disk.
- Simple uses would be hit-counters, real-time server reports, generating e-mail from web-based forms, etc.
- Compiled program executes quickly, and code can be kept elsewhere.

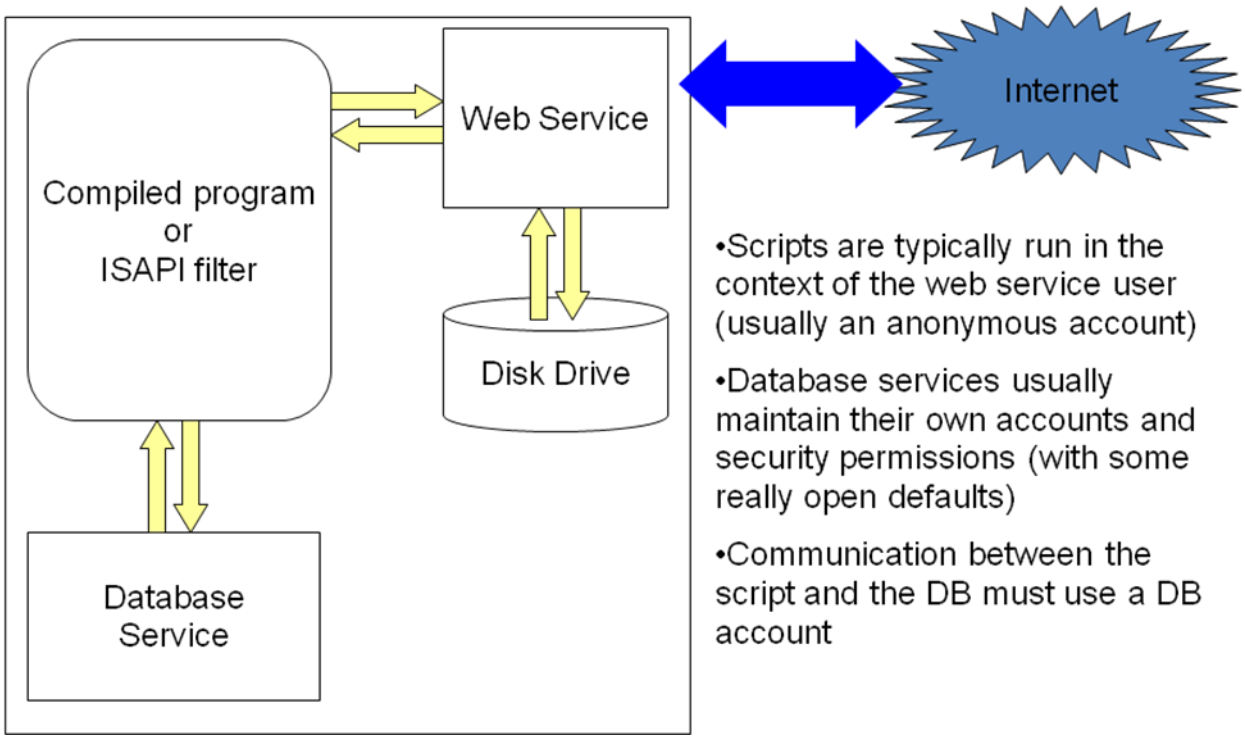


Web services

Programs saved as .DLL files

- Web service recognized hits to particular file types as requests for ISAPI-generated data.
- Used in MS's web-based server administration system.





Simple Voice Chat
A simple peer to peer voice chat application using sockets

Introduction

This is a simple voice chat program implemented in an ActiveX control (`OVoiceChatt` Control) using windows sockets in non compressed PCM format. You just need to give your name and the IP address of the computer on which you want to establish a voice chat session.

There is a simple test application (OVoiceChattClient) which has implemented the control.

Run the OVoiceChatClient.exe and enter you name and the ip address of the computer same application should be running on that computer as well. .A request for the voice chat goes to that computer and if that person accepts it then the voice chat starts.

To use in a program.

Below is some sample code that demonstrates using the control in your code.

In the header:

```
COVoiceChatt m_ctlVoice;
```

In the implementation:

```
BEGIN_EVENTSINK_MAP(COVoiceChattClientDlg, CDialog)
    //{{AFX_EVENTSINK_MAP(COVoiceChattClientDlg)
    ON_EVENT(COVoiceChattClientDlg, IDC_OVOICECHATTCTRL1, 1 /* GetVoiceInvitation */, \
            OnGetVoiceInvitation, VTS_BSTR VTS_BSTR)
    ON_EVENT(COVoiceChattClientDlg, IDC_OVOICECHATTCTRL1, 2 /* GetReqStatus */, \
            OnGetReqStatus, VTS_I4)
    ON_EVENT(COVoiceChattClientDlg, IDC_OVOICECHATTCTRL1, 3 /* GetVoiceEndNotice */, \
            OnGetVoiceEndNoticeOvoicechattctrl1, VTS_NONE)
    //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

void COVoiceChattClientDlg::OnGetVoiceInvitation(LPCTSTR ip, LPCTSTR nick)
{
    m_ip=ip;
    CString str=nick;
    str+=" wants to have voice chat with you";
    if(AfxMessageBox(str,MB_YESNO)==IDYES)
    {
        m_ctlVoice.OVoiceInvStatus(1,ip);
    }
    else
        m_ctlVoice.OVoiceInvStatus(0,ip);
}

void COVoiceChattClientDlg::OnGetReqStatus(long status)
{
    if(status==0)
        AfxMessageBox("request rejected");
    else
    {
        m_strStatus="Connecting";
        UpdateData(FALSE);
    }
}

void COVoiceChattClientDlg::OnButton1()
{
    m_ctlVoice.OVoiceInit();
}

void COVoiceChattClientDlg::OnButtonEnd()
{
    m_ctlVoice.OVoiceEnd();
}

void COVoiceChattClientDlg::OnGetVoiceEndNoticeOvoicechattctrl1()
{
    AfxMessageBox("Voice Conversation Has Been Ended");
}
```

Program to play mp3 and AVI files

Introduction

This program plays files of mp3 and avi format. This program is an illustration to use MCIWnd class.

Using the code

The main function or API is MCIWndCreate(). The MCIWndCreate function registers the MCIWnd window class and creates an MCIWnd window for using MCI services. MCIWndCreate can also open an MCI device or file (such as an AVI file) and associate it with the MCIWnd window.

```
HWND m_Video;
m_Video = NULL;
if(m_Video == NULL)
{
/*The MCIWndCreate function registers the MCIWnd window class
and creates an MCIWnd window for using MCI services. */


m_Video = MCIWndCreate(this->GetSafeHwnd(),AfxGetInstanceHandle(),
  WS_CHILD | WS_VISIBLE|MCIWNDF_NOMENU,m_Path);

}
else
{
    MCIWndHome(m_Video);    //go to the start
}
MCIWndPlay(m_Video);        //play the file
```

Similarly you can use MCIWndPause(m_Video) for pausing or MCIWndResume(m_Video) to resume the file.

```
BOOL Pause;
if(Pause)
{
  MCIWndResume(m_Video);
  Pause = FALSE;
}
else
{
  MCIWndPause(m_Video);
  Pause = TRUE;
}
```

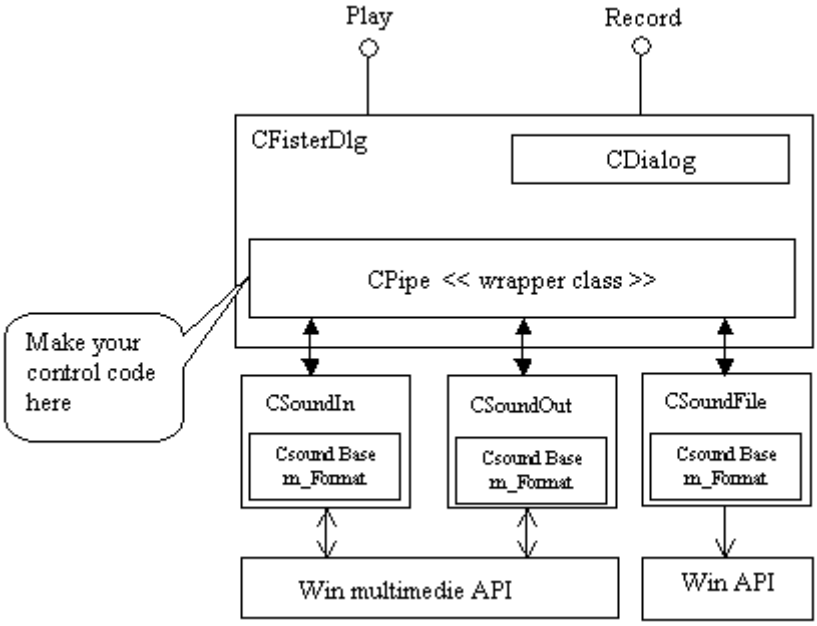
If you want to add more files viz. *.wav, *.mpeg, etc you can just append the extension to the fifth parameter of CFileDialog. CFileDialog box helps you to choose the file to be executed.

```
CFileDialog Cfile_dialog(TRUE,NULL,NULL,OFN_HIDEREADONLY,
  "MP3 Files (*.mp3)|*.mp3|AVI Files(*.avi)|*.avi|");
```

PLAYING MULTIMEDIA FILES

Introduction

This is a simple application that shows you how to play and record sound under windows.
It uses the old multimedia API. A better solution may be to use DirectSound.



Quick Guide to the Code

Start with the two functions in CFisterDlg called OnPlay and OnRecord. Follow them down to the depth you need to use the classes.

Short description

CSoundIn is a wrapper class that will let you retreive sound from the soundcard. The main functions are Start() and Stop()

CSoundOut is a wrapper class that will let you play sound on the soundcard. The main functions are Start() and Stop()

CSoundFile is a wrapper class of a single wave file, it can either be a file reader or a file writer object. See the constructor.

CSoundBase is a very small class that encapsulates the wave format.

CBuffer is a very small class that encapsulates a simple one dimentional buffer.

The project has a number of different callback functions:

- One callback function makes the Play button change it's label to stop when it has finished playing the file. CDialogDlg enherits CPipe and overloads a function that CPipe can call when it has finished playing the wave file.
- Another callback function make it possible for CSoundIn to callback to CPipe when it has filled the input buffer. Thus CPipe can give CSoundIn a new buffer to fill.
- A clone of the above principle is also used in CSoundOut, which enables it to callback to the owner when it is finished playing the sound in a given buffer.

UNIT – V
ADVANCED CONCEPTS

PART – A (2 MARKS)

1. List out the advantages of DBMS
2. Define SQL
3. List out the functions in CRecordset class
4. List out the ODBC elements
5. List out the MFC classes for DAO
6. Define Dynaset
7. Define snapshot
8. Define Threads
9. Define event
10. Define IP, UDP and TCP
11. Define WinSock
12. Define WinInet
13. Define IIS
14. Define ISAPI Server
15. List the advantages of WinInet over WinSock

PART – B

1. How the Worker and Main Thread communicate with each other (16)
2. Explain how ODBC database connectivity is done in VC++ with sample application (16)
3. Write down the WinSock Server and Client Program (16)
4. a. Explain in detail about ISAPI server extension DLL (8)
b. Explain in detail about MFC ISAPI server extension classes (8)
5. Write a program to play a audio and Video file (16)
6. Write a VC++ program to query the database (16)
7. Write a MFC automation client program (16)
8. Write a program to implement a WinInet Client using openURL (16)

Reg. No. : | | | | | | | | | | | |

T 3174

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2008.

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1253 — VISUAL PROGRAMMING

(Common to B.E. (Part-Time) Third Semester Regulation 2005)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — ($10 \times 2 = 20$ marks)

1. Explain in brief the windows MSG structure.

2. List the styles for a push button child window control.

3. State the use of windows message and event handler.

4. Give the resource script file for a modal dialog box.

5. State some messages and selection flags meant for a menu resource.

6. Explain in brief the MFC serialization mechanism.

7. What is an Active X control?

8. Define OLE.

9. State the MFC classes used for ODBC Database programming.

10. State the techniques used for playing audio files.

PART B — (5 × 16 = 80 marks)

11. (a) (i) Explain in detail the GDI Objects. (1) Device Context (2) Brush (3) Pen. (9)

(ii) Write a Win32 program to draw different shapes using different colors and fill styles. (7)

Or

(b) Explain the windows message map architecture in detail with a sample program. (16)

12. (a) (i) Describe in detail the Microsoft Foundation classes relating to user interfaces. (6)

(ii) Explain the CListBox class and its member Functions in detail with an example program. (10)

Or

(b) (i) State the use and explain the common dialog control in detail. (10)

(ii) Explain the modal and modeless dialog boxes. (6)

13. (a) (i) Write notes on the Rich Edit Control in detail. (6)

(ii) Explain how reading and writing is done in a SDI Document. (10)

Or

(b) What is DLL? Explain it types and advantages in detail. Write a program to create a user defined DLL. (16)

14. (a) Develop an application to create a simple Active X control using MFC.(16)

Or

(b) Explain the OLE drag and drop mechanism in detail. (16)

15. (a) (i) Explain the multithreading concept in VC++. (8)

(ii) Explain the use of WinInet API. (8)

Or

(b) Develop a database application for maintaining student details. (16)

———————

C 3146

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2007.

Fourth Semester

(Regulation 2004)

Computer Science and Engineering

CS 1253 — VISUAL PROGRAMMING

(Common to BE (Part-time) Third Semester Regulation 2005)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Define Registering of window class.

2. What are the events generating WM_PAINT message?

3. What are the mapping modes?

4. What are the device context classes?

5. What is the use of property sheets?

6. Write a serialize function.

7. Differentiate ActiveX controls and Ordinary windows controls.

8. What is OLE?

9. Mention any four differences between process and thread?

10. What is the function of IIS?

PART B — (5 × 16 = 80 marks)

11. (a) Explain Windows programming architecture with sample program.

Or

(b) Write a Windows program to create 3 scroll bars and change the windows background color using the scroll bar thumb positions.

12. (a) Explain VC++ components.

Or

(b) (i) Write a VC++ program to create a Modal and modeless dialog boxes.

(ii) Differentiate modal and modeless dialog.

13. (a) Write a VC++ program to replace the standard application framework Edit, Cut, Copy, and paste toolbar buttons with 3 special-purpose buttons that control drawing in the view window.

Or

(b) (i) Compare the features of SDI and MDI applications with sample code.

(ii) With neat diagram discuss the hierarchy of view Classes supported in MFC.

14. (a) Explain the creation and usage of ActiveX control.

Or

(b) Explain (i) Containment (ii) Aggregation (iii) Inheritance with suitable example.

15. (a) Write a program in MFC to create a worker thread and explain the following:

(i) Controlling the thread from the main thread.

(ii) Sending messages between Main and Worker thread.

Or

(b) Write a program to query the database.

———————