

## References:

*“Introduction to MFC”*, Deitel, Deitel, Nieto & Strassberger, Prentice Hall © 2000

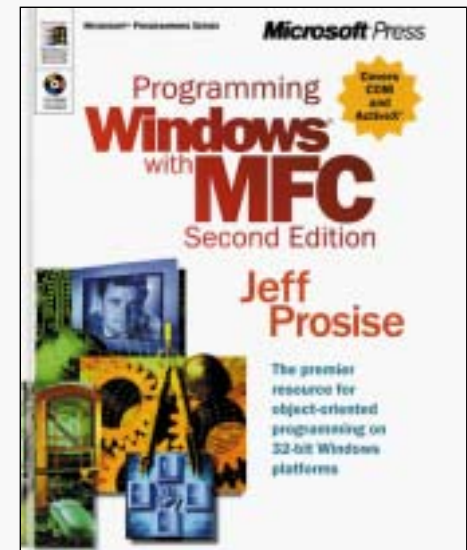
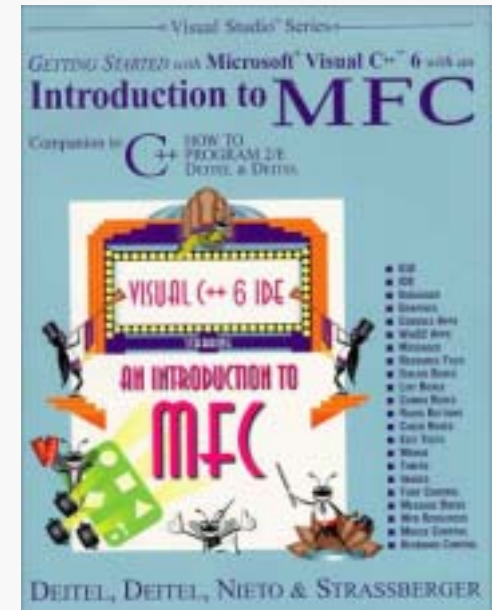
<http://vig.prenhall.com/catalog/academic/product/1,4096,0130132497,00.html>

<http://www.amazon.com/exec/obidos/ASIN/0130161470/deitelassociatin>

*“Programming Windows® with MFC”*, Jeff Prosise, Microsoft Press © 1999

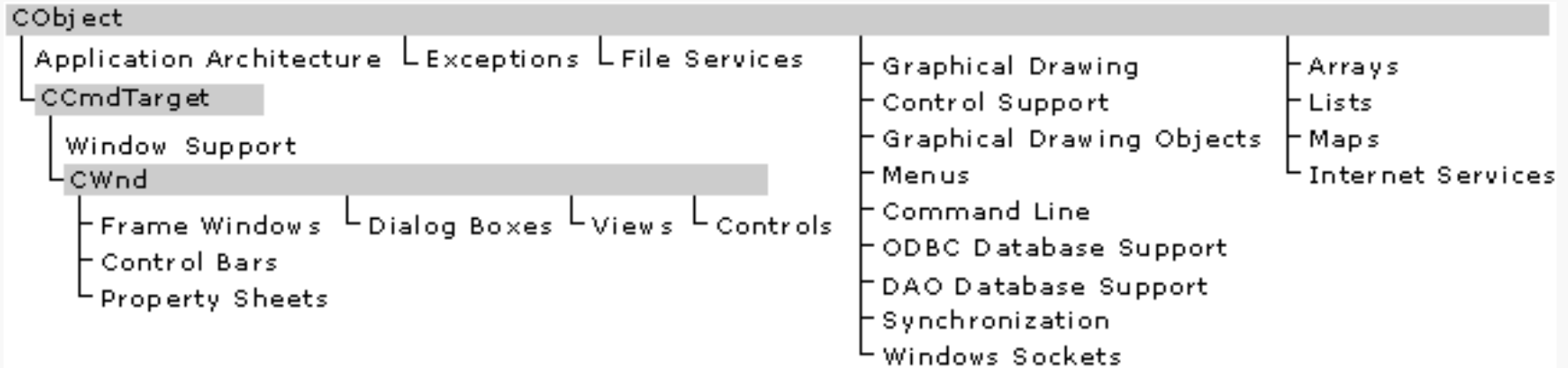
<http://www.microsoft.com/mspress/books/1431.asp>

[http://www.amazon.com/exec/obidos/ASIN/1572316950/qid=1007995908/sr=1-1/ref=sr\\_1\\_15\\_1/103-8969452-5438253](http://www.amazon.com/exec/obidos/ASIN/1572316950/qid=1007995908/sr=1-1/ref=sr_1_15_1/103-8969452-5438253)



## MFC Class Hierarchy

Common window/GUI behaviors implemented in base classes for derived classes to inherit and over-ride for specific application behavior.



Only a small subset of the MFC class hierarchy will be covered herein.

**CWinApp**: a base class for creating & executing Win applications

**CDialog**: a base class for creating & managing dialog windows

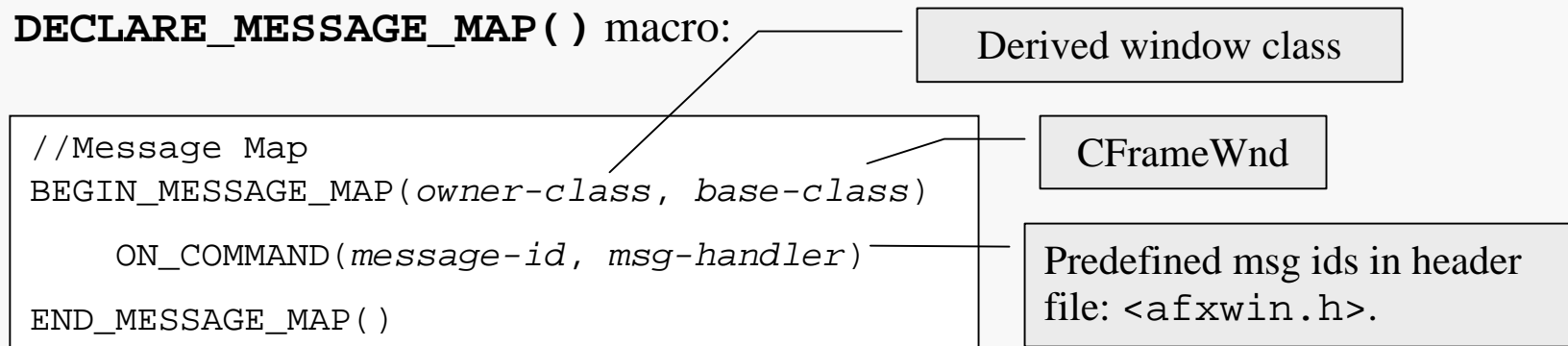
**CFrameWnd**: a base class for creating & managing frame windows

## GUI Events

GUI programs are event driven. Applications responds to user actions which generates events, (mouse, keyboard, etc.). Events are communicated to a program through messages sent by the Op Sys. Program processing characterized by an event-loop which receives and responds to messages. MFC provides the necessary event-loop logic in an object-oriented application framework.

## Message Handling

A MFC application, (app) derives classes from the MFC hierarchy that contain code to react to events (**message handlers**). Every message, (msg), is specified by a unique int, (**message identifier**). Message handlers are associated with message identifiers through a **message map**. The map registers handlers with the event-loop. When the app receives a msg it looks up the identifier to find the corresponding handler to execute for the appropriate response. The map is defined by the **DECLARE\_MESSAGE\_MAP ( )** macro:

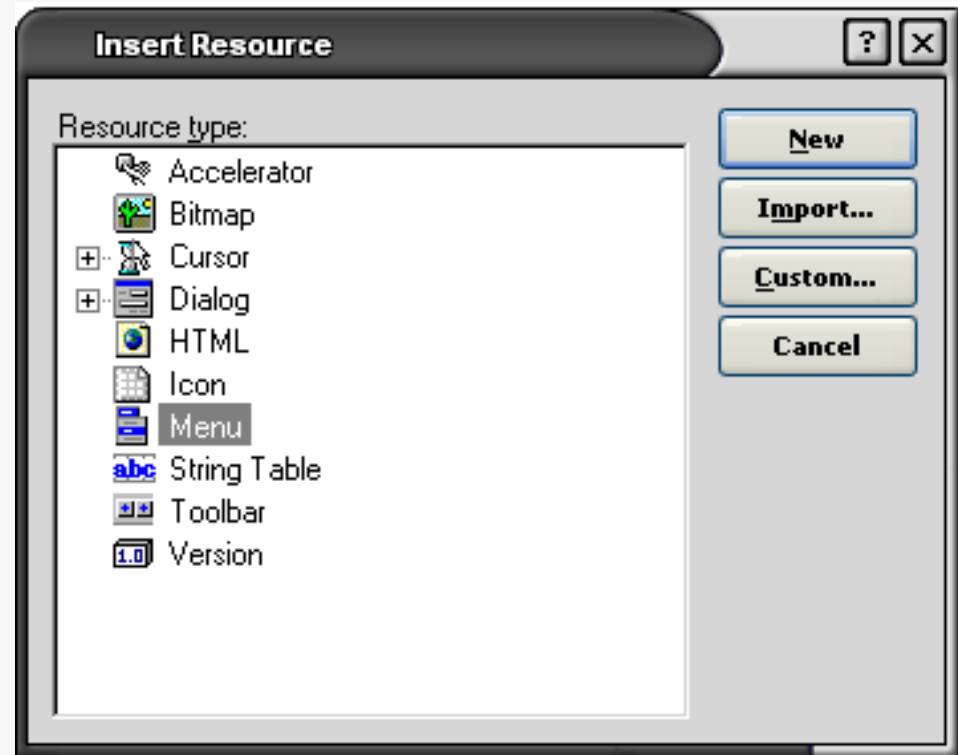


## Resource Definitions

MSVC++ supports a resource definition language, (RDL), for the specification of GUI controls: (type, location, size, msg id, etc.).

RDL statements stored in a `resource.rc` file in a Win32 application project.

Resource files can be created & edited by hand, but usually the IDE resource editor is used to graphically design the interface controls. A resource compiler translates `resource.rc` files into object code.



## Variable Prefix Naming Convention

To more easily identify type and scope, (w/o having to refer back to the definition), most MFC programmers employ Hungarian notation

<b>Prefix</b>	<b>Data type</b>
ar	array
b	Boolean
c	Char
C	Class
dw	DWORD, double word or unsigned long
fn	Function
h	Handle
i	int (integer)
m	member
n	short int
p	a pointer variable containing the address of a variable
s	string
sz	ASCIIZ null-terminated string
s_	static class member variable
w	WORD unsigned int

## Win32 Project

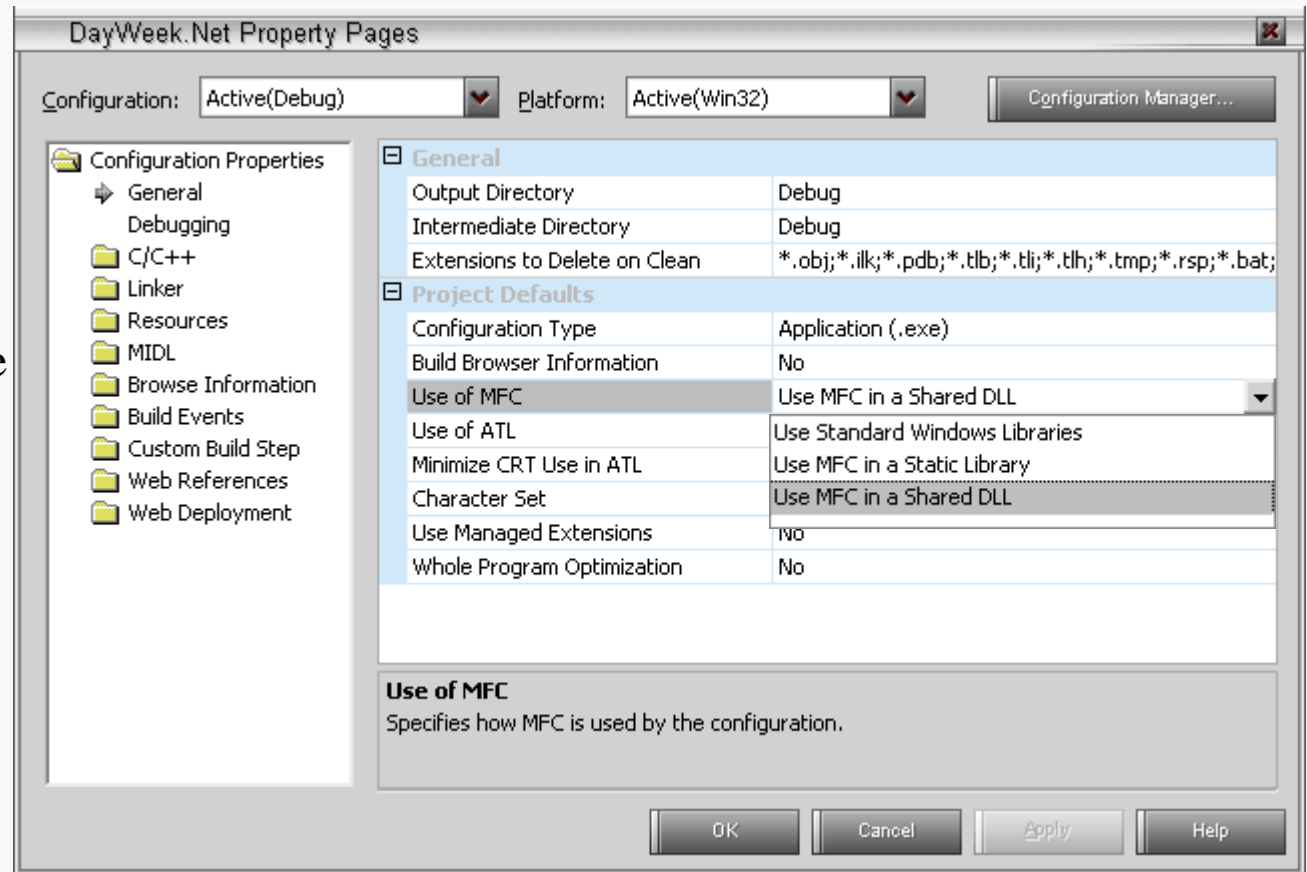
In Visual Studio, create an empty Win32 Project and choose “Windows application”.

## MFC library

Select menu/option: Project / Properties

For the “Use of MFC” entry, select “Use MFC in a Shared DLL.”

Linking to the MFC  
DLL decreases exe size  
& compilation time.



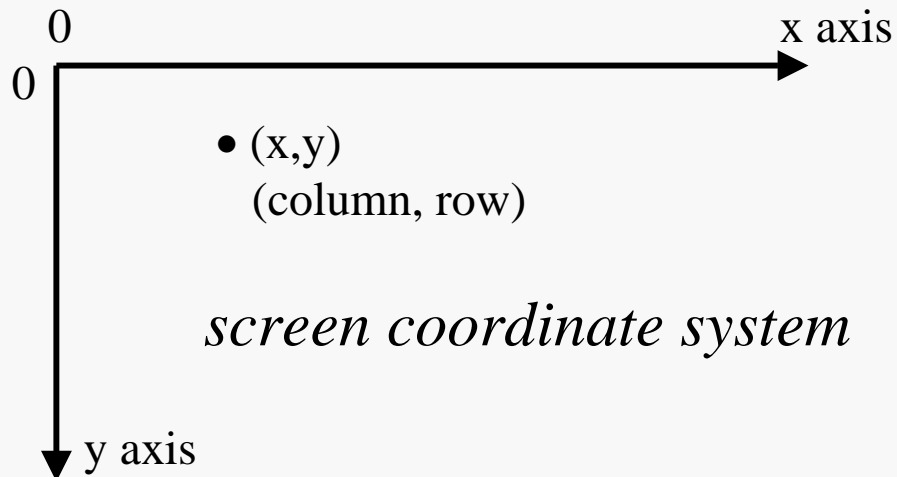
## Code

```
1. //HelloWorld.h
2. class CHelloWindow : public CFrameWnd {
3. public:
4.     CHelloWindow();    // constructor initializes window
5.     ~CHelloWindow();  // destructor releases resources

6. private:
7.     CString m_Hello;  // contains Hello World string
8. };
```

Line 4. derives from CFrameWnd inheriting basic window functionality.

Line 7. Defines a MFC CString object.



```
9. //HelloWorld.cpp
10. // include application framework windows class definitions:
11. #include <afxwin.h>           // application frameworks header
12. #include "HelloWorld.h"      // class definition for application

13. // constructor initializes the window
14. CHelloWindow::CHelloWindow()
15. {
16.     // Create Window with Title Bar
17.     Create( NULL,              // default CFrameWnd class
18.            "Hello",           // window title
19.            WS_OVERLAPPEDWINDOW, // full-featured window
20.            CRect( 200, 100, 350, 200 ) ); // screen coordinates

21.     m_Hello.Create(           // create Windows control
22.        "Hello World",        // text
23.        WS_CHILD | WS_VISIBLE | WS_BORDER // window styles
24.        | SS_CENTER,          // static object styles
25.        CRect( 20, 30, 120, 50 ), // window coordinates
26.        this );               // context that owns child window
27. }
```

```
29. CHelloWindow::~CHelloWindow()
30. {
31. }

32. // derive application class from CWinApp
33. class CHelloApp : public CWinApp {
34. public:
35.     BOOL InitInstance()    // override default function
36.     {
37.         m_pMainWnd = new CHelloWindow();    // create window
38.         m_pMainWnd->ShowWindow( m_nCmdShow ); // make visible
39.         m_pMainWnd->UpdateWindow();         // force refresh
40.         return TRUE;                       // report success
41.     }

42. } HelloApp;    // instantiate application
```

## Window Creation

Line 11. (`#include <afxwin.h>` // application frameworks header)

Includes standard MFC message Ids, handlers and map.

Lines 17-20:

```
17. Create( NULL,                // default CFrameWnd class
18.         "Hello",            // window title
19.         WS_OVERLAPPEDWINDOW, // full-featured window
20.         CRect( 200, 100, 350, 200 ) ); // screen coordinates
```

Creates the main application window. The `NULL` argument instructs Windows to use the default window properties for this window class. The `WS_OVERLAPPEDWINDOW` setting creates a resizable window with standard window controls. The last `Create()` argument instantiates a `CRect`(rectangle) object to store the window screen coordinates. The first (x,y) pair gives the top-left coordinate and the last (x,y) pair gives the lower-right coordinate. This defines a window that is 150 x 100 pixels, (width, height).

## CStatic creation

Lines 21-27: creates a `CStatic` object (used for text labels).

```
21. m_Hello.Create(           // create Windows control
22.     "Hello World",       // text
23.     WS_CHILD | WS_VISIBLE | WS_BORDER // window styles
24.     | SS_CENTER,         // static object styles
25.     CRect( 20, 30, 120, 50 ), // window coordinates
26.     this );              // context that owns child window
27. }
```

The first argument is the text label to be displayed. The second argument is mask to set the `CStatic` window characteristics. It is formed by logically OR'ing together pre-defined window style, (`WS`), constants: (`WS_CHILD` : sub-window; `WS_VISIBLE` : viewable; `WS_BORDER` : rectangular border; `SS_CENTER` : center text).

Line 26: gives the owner (parent) argument for the `CStatic` subwindow, `this` window is the `CHelloWindow` window, (establishes an association between the sub-window and parent window, which allows the MS Win OS to move the window in memory).

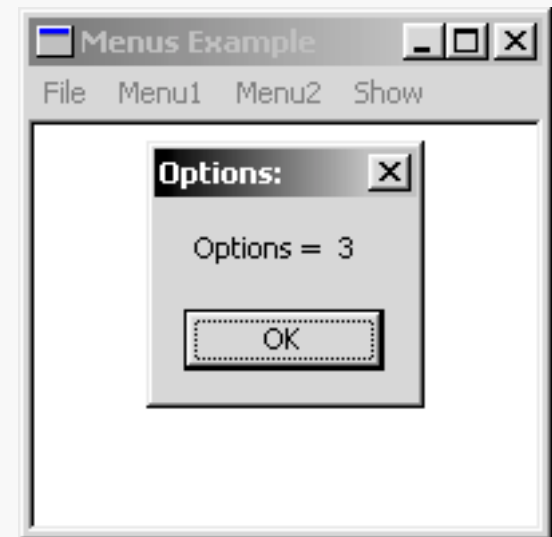
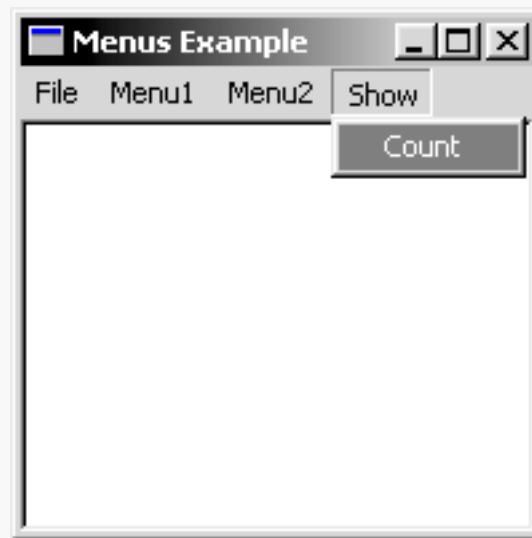
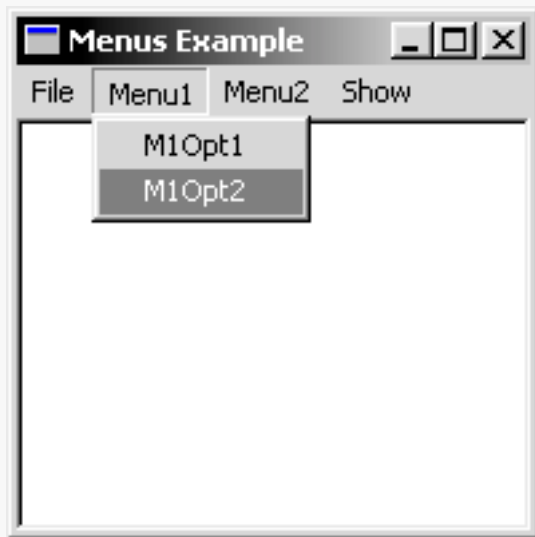
## Win Application Class: Lines 33-42:

```
32. // derive application class from CWinApp
33. class CHelloApp : public CWinApp {
34. public:
35.     BOOL InitInstance()    // override default function
36.     {
37.         m_pMainWnd = new CHelloWindow();    // create window
38.         m_pMainWnd->ShowWindow( m_nCmdShow ); // make visible
39.         m_pMainWnd->UpdateWindow();    // force refresh
40.         return TRUE;    // report success
41.     }
42. } HelloApp;    // instantiate application
```

All MFC app's must have 1 (& only 1) instance of a class derived from CWinApp. The CWinApp class controls application instantiation, execution (event loop), and destruction. The main() is replaced is replaced by CWinApp. InitInstance() instantiates the app main window object and begins execution. The <x>Window() FNs above are inherited from CWinApp. MFC apps (must) use a pre-defined BOOL (int) type with TRUE/FALSE constants instead of the standard C++ bool type. The m\_nCmdShow inherited variable indicates that the win is to be initially displayed unmaximized and unminimized.

A trivial menu options counter application.

The following MFC code displays a window with a few menus, allowing a user to select options from Menu1 and Menu2. The Show menu count option then displays a count of the number of options selected from the preceding two menus in a message dialog box.



## Code

```
1.  // WinMenus.h
2.  // create menus with MFC
3.
4.  class CMenuWin : public CFrameWnd {
5.  public:
6.      CMenuWin();
7.      afx_msg void OnExit();
8.      afx_msg void OnCount();
9.      afx_msg void OnShowCount();
10. private:
11.     int m_iTotal;           // count menu options selected
12.     ostream m_str;        // output string stream
13.     DECLARE_MESSAGE_MAP()
14. };
```

Menu “callback” FNs.  
Invoked when menu options  
are selected.

Invokes the message map  
macro to map message IDs  
to the message handler FNs.

```
15. // WinMenus.cpp
16. // create simple menus with MFC
17. #include <afxwin.h>           // MFC application framework
18. #include <strstream.h>       // C-style string stream class
19. #include <iomanip.h>         // I/O manipulators
20. #include "WinMenusIDs.h"    // application message ID symbols
21. #include "WinMenus.h"

22. CMenuWin::CMenuWin()        // construct window
23. {
24.     Create( NULL, "Menus Example", WS_OVERLAPPEDWINDOW,
25.           CRect( 100, 100, 300, 300 ), NULL, "Count" );

26.     m_iTotal = 0;
27. }
```

- The `CRect` constructor could have been replaced by the MFC pre-defined `CRect` object `rectDefault`, to allow Windows to choose the initial size and placement.
- The second `NULL` argument indicates that this is a root window having no parent.
- The `"Count"` argument associates the menu defined in the resource file with the window.

```
28. // afx_msg precedes each message handler function
29. afx_msg void CMenuWin::OnExit() ←
30. {
31.     SendMessage( WM_CLOSE );
32. }

33. // count each menu option selected
34. void CMenuWin::OnCount( ) ←
35. {
36.     m_iTotal++;
37. }

38. afx_msg void CMenuWin::OnShowCount( )
39. {
40.     m_str.seekp( 0 ); // reset output string stream
41.     m_str << setprecision( 2 )
42.         << setiosflags( ios::fixed | ios::showpoint )
43.         << "Options = " << m_iTotal << ends; // stopper

44.     // display new dialog box with output string
45.     MessageBox(m_str.str(), "Options:" );
46. }
```

afx\_msg is the MFC prefix used to mark a msg handler. The WM\_CLOSE msg terminates execution.

Msg handler FN for all Menu1 & Menu2 options, to update the option selection counter. It receives & handles a range of msg Ids.

Returns a C-style string (char \*) from the ostream m\_str object.

MessageBox FN displays a popup msg dialog window. It accepts a C-style string to display and a dialog win label string.

```
47. BEGIN_MESSAGE_MAP( CMenuWin, CFrameWnd )
48.     ON_COMMAND( IDM_EXIT, OnExit )
49.     ON_COMMAND_RANGE( IDM_M101, IDM_M203, OnCount )
50.     ON_COMMAND( IDM_SHOW_COUNT, OnShowCount )
51. END_MESSAGE_MAP()

52. class CMenuApp : public CWinApp {
53. public:
54.     BOOL InitInstance()           // called by CWinApp::CWinApp
55.     {
56.         m_pMainWnd = new CMenuWin;           // create window
57.         m_pMainWnd->ShowWindow( m_nCmdShow ); // make it visible
58.         m_pMainWnd->UpdateWindow();          // force refresh
59.         return TRUE;                         // report success
60.     }
61. } menuApp;                                // calls CWinApp::CWinApp
```

Invokes msg map macro to associate msg Ids with handler FNs.

IDM\_ prefix indicates an identifier of a menu using MFC naming conventions. Msg handlers have the prefix On.

Derives the app class from `CWinApp` and instantiates it.

## Message Identifiers

Predefined MFC message identifiers are in the range: [0 ... 1023]

Programmer-defined message identifiers are in the range: [1024 ... 65535]

```
63. // WinMenuIDs.h
64. // define messages used by menus.cpp and menus.rc
65. #define   IDM_EXIT           2000 ← File/Exit msg id
66. #define   IDM_M101          2011 ← Menu1 msg ids
67. #define   IDM_M102          2012
68. #define   IDM_M201          2021 ← Menu2 msg ids
69. #define   IDM_M202          2022
70. #define   IDM_M203          2023
71. #define   IDM_SHOW_COUNT    2031 ← Show/Count msg id
```

Message Ids support the connections between the messages and associated handlers.

```
72. // WinMenus.rc
73. // resource script for menus example
74. #include <afxres.h>
75. #include "WinMenusIDs.h"

76. Count MENU
77. {
78.     POPUP "File"
79.     {
80.         MENUITEM "Exit", IDM_EXIT
81.     }

82.     POPUP "Menu1"
83.     {
84.         MENUITEM "M1Opt1", IDM_M101
85.         MENUITEM "M1Opt2", IDM_M102
86.     }

87.     POPUP "Menu2"
88.     {
89.         MENUITEM "M2Opt1", IDM_M201
90.         MENUITEM "M2Opt2", IDM_M202
91.         MENUITEM "M2Opt3", IDM_M203
92.     }
```

```
93. POPUP "Show"
94.     {
95.         MENUITEM "Count", IDM_SHOW_COUNT
96.     }
97. }
```

WinMenus.rc resource file defines the menu and options to msg Id associations. The lines are resource definition statements, (the MS Win GUI description language). Can be created in a text editor and added to the project.

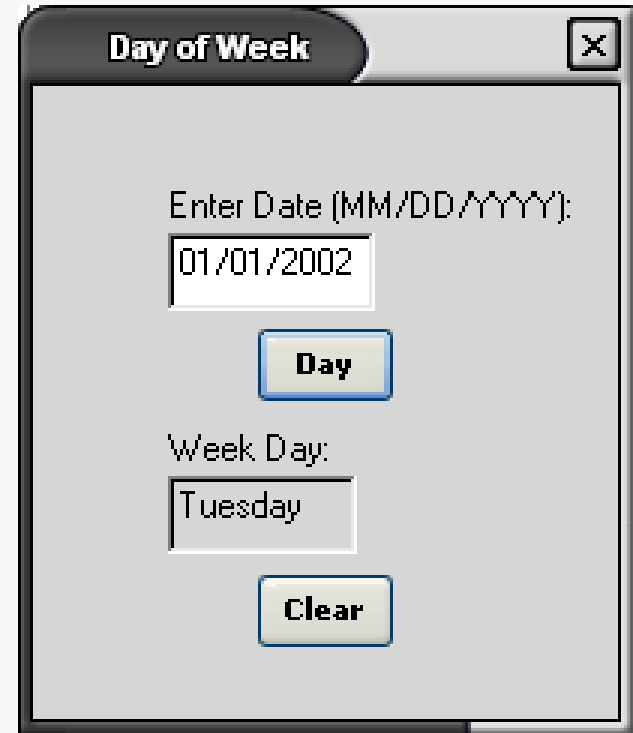
Note: creating a resource file within the MS VC IDE and opening it will invoke the graphical resource editor which is not covered in these notes.

## Dialog Based Application

The code on the following slides discusses the code for simple MFC dialog window based application. The program allows a user to enter a date for the Gregorian calendar and displays the corresponding day of the week for the date.

The code introduces dialog boxes, button controls and edit text/box controls.

Error checking for valid Gregorian calendar dates is not included to focus on the MFC code. The application will currently accept invalid dates and incorrectly formatted user input.



```
// DayWeek_ids.h
// Define Message Numbers

#define IDC_DATE    2000
#define IDC_DAY     2001
#define IDC_WEEK    2002
#define IDC_CLEAR   2003
```

Defines the dialog controls message Ids.

```
1. // DayWeek.h
2. // Day of Week MFC dialog program
3. class CDayWeek : public CDialog {
4. public:
5.     CDayWeek()
6.         : CDialog( "DayWeek" ), m_nDay(1), m_nMon(1), m_nYear(2000)
7.     {}
8.
9.     afx_msg void OnDay();           // clicked the "Day" button
10.    afx_msg void OnClear();         // clicked the "Clear" button
11.
12. private:
13.     int m_nDay, m_nMon, m_nYear;   // Date
14.
15.     DECLARE_MESSAGE_MAP()
16. };
```

Dialog resource "DayWeek" is sent to inherited CDialog constructor to pass dialog attributes.

Dialog box button "callback" FNs. Invoked when buttons are clicked.

```
13. // DayWeek.cpp
14. // Day of Week MFC dialog program
15. #include <afxwin.h>
16. #include <strstream.h> // C-string streams
17. #include <string>
18. #include "DayWeek_ids.h"
19. #include "DayWeek.h"

20. // clicked the "Day" button
21. afx_msg void CDayWeek::OnDay()
22. {
23.     const int TEXT_SIZE = 11;
24.     char tmp, szText[ TEXT_SIZE + 1 ]; // buffer for conversions

25.     // get addresses of Edit Box Controls
26.     CEdit *pDate = ( CEdit * ) ( GetDlgItem( IDC_DATE ) );
27.     CEdit *pWeek = ( CEdit * ) ( GetDlgItem( IDC_WEEK ) );

28.     pDate->GetWindowText( szText, TEXT_SIZE ); // get Date
29.     std::string DayOfWeek(szText); //initialize string
```

GetDlgItem() returns base type (CWnd\*) pointers, which are type cast to the derived pointer type, (CEdit \*).

GetWindowText() returns the dialog edit box control text, (C-style string), using the dialog control pointer.

Lines 26-27: get the addresses of the edit dialog boxes for manipulation. The IDC\_ codes, defined in DayWeek\_ids.h, are passed to GetDlgItem() for the address lookup.

Code to parse input and compute day of week.

```
30. if (DayOfWeek[0] == '0') DayOfWeek.erase(0,1); //del leading day zero
31. int m1 = DayOfWeek.find("/",0) + 1; //find start of month char position
32. if (DayOfWeek[m1] == '0') DayOfWeek.erase(m1,1); //del leading mon zero

33. istrstream istr((char*) DayOfWeek.c_str()); //init input istrstream
34. istr >> m_nMon >> tmp >> m_nDay >> tmp >> m_nYear; //read date

35. if (m_nMon < 3) { //formula requires Jan & Feb
36.     m_nMon += 12; //be computed as the 13th & 14th months
37.     m_nYear -= 1; //of the preceding year
38. } //if

39. switch ( (m_nDay + 2 * m_nMon + 3 * (m_nMon + 1) / 5 + m_nYear
40.     + m_nYear / 4 - m_nYear / 100 + m_nYear / 400 + 1) % 7) {
41.     case 0: DayOfWeek = "Sunday"; break;
42.     case 1: DayOfWeek = "Monday"; break;
43.     case 2: DayOfWeek = "Tuesday"; break;
44.     case 3: DayOfWeek = "Wednesday"; break;
45.     case 4: DayOfWeek = "Thursday"; break;
46.     case 5: DayOfWeek = "Friday"; break;
47.     case 6: DayOfWeek = "Saturday"; break;
48.     default: DayOfWeek = "Error";
49. } //switch
```

```
50.     pWeek->SetWindowText( DayOfWeek.c_str() ); // display week day
51.     pDate->SetFocus(); // next date
52. } // OnDay()
```

SetWindowText() sets the contents of the dialog text box, using the dialog control pointer. SetFocus() makes the control *active*, (ie, user doesn't need to click it to edit the date text).

```
50. // clicked the "Clear" button
51.	afx_msg void CDayWeek::OnClear()
52. {
53.     // get addresses of Edit Box Controls
54.     CEdit *pDate = ( CEdit * ) ( GetDlgItem( IDC_DATE ) );
55.     CEdit *pWeek = ( CEdit * ) ( GetDlgItem( IDC_WEEK ) );

56.     m_nDay = m_nMon = 1; m_nYear = 2000;

57.     pDate->SetWindowText( "" ); // clear the date edit box
58.     pWeek->SetWindowText( "" ); // clear the week day box
59.     pDate->SetFocus(); // next date to input
60. } // OnClear()
```

The `GetDlgItem()` FN must be re-called every time to manipulate the controls, (because the OS reallocates memory every time Windows are created and destroyed).

```
64. BEGIN_MESSAGE_MAP( CDayWeek, CDialog )
65.     ON_COMMAND( IDC_DAY, OnDay )
66.     ON_COMMAND( IDC_CLEAR, OnClear )
67. END_MESSAGE_MAP()

68. // dialog-based application
69. class CDayWeekApp : public CWinApp {
70. public:
71.     BOOL InitInstance()
72.     {
73.         CDayWeek DayWeekDialog;
74.         DayWeekDialog.DoModal(); // run dialog
75.         return FALSE;           // finished
76.     }

77. } DayWeek;
```

The message macro maps the dialog button message IDs to the button handler FNs.

Derives the app class from `CWinApp` and instantiates it. The dialog window is instantiated and the `DoModal()` is invoked to display it as *modal* window, (modal windows require the user to respond to them before anything else can be done).

```

78. // DayWeek.rc
79. // resource script for DayWeek
80. #include "afxres.h"
81. #include "DayWeek_ids.h"

82. DayWeek DIALOG 50, 50, 130, 130
83. STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU

84. CAPTION "Day of Week"
85. {
86.     LTEXT      "Enter Date (MM/DD/YYYY):", IDC_STATIC, 30, 20, 98, 8
87.     EDITTEXT   IDC_DATE, 30, 30, 46, 16, ES_AUTOHSCROLL

88.     DEFPUSHBUTTON "Day", IDC_DAY, 50, 50, 30, 15

89.     LTEXT      "Week Day:", IDC_STATIC, 30, 70, 50, 8
90.     EDITTEXT   IDC_WEEK, 30, 80, 42, 16,
91.     ES_READONLY | NOT WS_TABSTOP

92.     PUSHBUTTON "Clear", IDC_CLEAR, 50, 100, 30, 15,
93.     NOT WS_TABSTOP
94. }

```

Dialog win location: upper left corner is at (50,50) pixels. The last 2 numbers give horizontal, vertical dialog unit size. Horizontal units = 0.25 char width, vertical = 0.125 char height.

Dialog box title.

Dialog styles prefixed with DS\_.

Edit style allows horizontal scrolling w/o scroll bar.

Defines a output control w/o focus, (when user hits tab key control is skipped).

IDC\_STATIC controls do not generate msgs, (does not require control IDC\_number). LTEXT is a left aligned control, (RTEXT, CTEXT also available). WS\_SYSMENU includes a system win menu with Move & Close options (right-click title bar). WS\_POPUP wins are parentless.

## Types of Controls

The Windows environment provides several types of controls a programmer can use to design a user interface. The following types are the most common:

- Buttons
- Static Controls
- Check Boxes
- Radio Buttons
- Edit Boxes
- List Boxes
- Combo Boxes

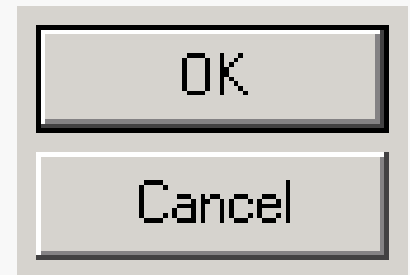
Other more advanced types of controls that Windows and MFC offer in recent versions include hierarchical tree views, iconic list views, date/time pickers, tabbed property sheets, and many others. Since these notes are only meant to provide an introduction, we will limit discussion to the controls listed above.

## Buttons

A button represents a control that a user clicks in order to carry out a specific action. It has no state—it only responds when clicked by notifying the parent window so the application can take an appropriate action.

The Windows button control is represented in MFC by the `CButton` class. In a Windows resource file, the statements `PUSHBUTTON` and `DEFPUSHBUTTON` are used to define them.

As an MFC programmer, you will rarely deal with the `CButton` class directly for simple push-buttons, because there is no state that requires being accessed. The event mapping mechanism is used to handle the `ON_COMMAND` notifications that are sent when a button is clicked, and the event handler function that you write will be invoked, similar to the menu handling mechanism discussed above.



```
1. // resource.h
2. #define IDD_BUTTONDLG          101
3. #define ID_HELLO                1000
```

```
1. // Button.rc
2. #include <afxres.h>
3. #include "resource.h"
4. IDD_BUTTONDLG DIALOG  0, 0, 76, 39
5. CAPTION "Button Dialog"
6. {
7.     DEFPUSHBUTTON    "Hello...", ID_HELLO, 13, 13, 50, 14
8. }
```

This resource script generates the dialog shown to the right.


The `DEFPUSHBUTTON` statement tells Windows to make this the default button—it will be rendered with a thin black border, and pressing Enter when the dialog is active will carry out the associated action as if it had been clicked. To create a standard non-default button, use the `PUSHBUTTON` statement instead.



```
1. // Button.h
2. class CButtonDlg : public CDialog
3. {
4. public:
5.     CButtonDlg();
6.
7.     // Called when "Hello..." is clicked
8.     afx_msg void OnHello();
9.
10. private:
11.     DECLARE_MESSAGE_MAP()
12. };
```

```
1. // Button.cpp
2. #include <afxwin.h>
3. #include "resource.h"
4. #include "Button.h"
5.
6. // Dialog constructor
7. CButtonDlg::CButtonDlg() : CDialog(IDD_BUTTONDLG) { }
8.
9. afx_msg void CButtonDlg::OnHello()
10. {
11.     // Display a message in response to the click.
12.     AfxMessageBox("...world!");
13. }
```

A dialog resource ID can also be a numeric constant defined in the resource files instead of a string literal



```
12. // For a more involved application, you could create a menu with the
13. // same command ID, and the same ON_COMMAND entry in the message map
14. // would take care of both.
15. BEGIN_MESSAGE_MAP(CButtonDlg, CDialog)
16.     ON_COMMAND(ID_HELLO, OnHello)
17. END_MESSAGE_MAP()

18. class CButtonApp : public CWinApp
19. {
20. public:
21.     BOOL InitInstance()
22.     {
23.         // Create and show the dialog.
24.         CButtonDlg dlg;
25.         dlg.DoModal();
26.         return FALSE;
27.     }

28. } buttonApp;
```

ON\_COMMAND handler associated with the button ID. OnHello() will be called when the button with this ID is clicked

When the user clicks the “Hello...” button, MFC will search the message map for an entry associated with `ID_HELLO`. In our example, it will find the function `OnHello()` and execute it.

`AfxMessageBox()` is a global MFC function used to display a simple message dialog on the screen with the specified text prompt.



## Static Controls

Static controls are used to label other controls that do not have labels of their own, such as edit boxes and list boxes. They accept no user input, and thus are not used in message maps.

Since static controls typically neither accept input nor need to be referenced after they are created, a standard resource ID can be associated with them. The constant `IDC_STATIC` is defined as the value `-1` in `<winres.h>` (which is automatically included by `<afxwin.h>`).

To create a static control in a dialog, use a resource statement similar to the following:

```
LTEXT          "Static", IDC_STATIC, 5, 71, 19, 8
```

The last four numbers represent the location and size of the control. The type `LTEXT` indicates that the text is left-aligned. Other types available are `RTEXT` for right-aligned text, and `CTEXT` for centered text.

## Check Box Controls

A check box is a control that allows the user to specify a Boolean true/false value for the state of some object in the application.

There is also third state, “indeterminate”, available on three-state checkboxes, which is used in situations where multiple objects are selected and a given property of those objects has different values.

For example, in a word processor, if the user selected a range of text in which part of the selection is bold and another part is not, the “bold” button would be displayed in the “indeterminate” state.

This discussion will be limited to Boolean (two-state) check boxes.

Like standard push buttons, check boxes are represented by the `CButton` MFC class.



## Check Box Resource Statements

The following resource statements are available to define a check box in a dialog resource:

- **AUTO3STATE**: A three-state check box that changes state when clicked.
- **AUTOCHECKBOX**: A Boolean check box that changes state when clicked.
- **CHECKBOX**: A Boolean check box that does not change state when clicked.
- **STATE3**: A three-state check box that does not change state when clicked.

You will almost always use the **AUTO-** versions of the statements, because they switch between their possible states automatically when clicked. The non-**AUTO-** versions do not change state on their own, but can send notifications to the parent dialog when they have been clicked. You can then decide to allow the change and check/uncheck the control manually, but this is not commonly done.

```
1. // resource.h
2. #define IDD_CHECKBOXDLG          101
3. #define IDC_CS1044                1001
4. #define IDC_CS1704                1002
5. #define IDC_CS2704                1003
6. #define IDC_CHECK_STATES          1004
```

```
1. // CheckBox.rc

2. #include <afxres.h>
3. #include "resource.h"

4. IDD_CHECKBOXDLG DIALOG 0, 0, 104, 86
5. CAPTION "Check Box Dialog"
6. {
7.     DEFPUSHBUTTON    "Check States", IDC_CHECK_STATES, 48, 65, 50, 14
8.     AUTOCHECKBOX     "CS 1044", IDC_CS1044, 14, 17, 45, 10
9.     AUTOCHECKBOX     "CS 1704", IDC_CS1704, 14, 30, 45, 10
10.    AUTOCHECKBOX     "CS 2704", IDC_CS2704, 14, 43, 45, 10
11.    GROUPBOX         "Prerequisites", IDC_STATIC, 5, 5, 93, 55
12. }
```

The GROUPBOX statement is used to create a control that visually groups a set of controls, labeled with the specified title and drawn with a 3D etched border. Since it does not accept user input, IDC\_STATIC is typically used as its resource identifier.

```
1.  // CheckBox.h
2.  class CCheckBoxDlg : public CDialog
3.  {
4.  public:
5.      CCheckBoxDlg();

6.      // Called when "OK" is clicked
7.      afx_msg void OnCheckStates();

8.  private:
9.      DECLARE_MESSAGE_MAP()
10. };
```

```
1.  // CheckBox.cpp
2.  #include <afxwin.h>
3.  #include "resource.h"
4.  #include "CheckBox.h"

5.  // Dialog constructor
6.  CCheckBoxDlg::CCheckBoxDlg() : CDialog(IDD_CHECKBOXDLG) { }

7.  void AddCheckStateToString(CButton* pCheckBox, CString& str)
8.  {
9.      // Retrieves the state of a checkbox and concatenates
10.     // the string "checked" or "unchecked" and a newline to
11.     // the end of the string passed to the function.
```

```
12.     if(pCheckBox->GetCheck() == BST_CHECKED)
13.         str += "checked\n";
14.     else
15.         str += "unchecked\n";
16. }

17.	afx_msg void CCheckBoxDlg::OnCheckStates()
18.	{
19.		// Display a message box showing the checked states
20.		// of the three checkboxes in the dialog.
21.		CString str;

22.		CButton* pCS1044 = (CButton*)GetDlgItem(IDC_CS1044);
23.		CButton* pCS1704 = (CButton*)GetDlgItem(IDC_CS1704);
24.		CButton* pCS2704 = (CButton*)GetDlgItem(IDC_CS2704);
```

The `AddCheckStateToString()` function uses `CButton::GetCheck()` to determine the state of the check box. `GetCheck()` returns an integer, which is one of the following predefined values:

- `BST_UNCHECKED`: The check box is unchecked.
- `BST_CHECKED`: The check box is checked.
- `BST_INDETERMINATE`: For a 3-state check box, the check box is in the indeterminate state.

```
25.     // Build the string that will be displayed in the
26.     // message box.
27.     str += "CS 1044 is ";
28.     AddCheckStateToString(pCS1044, str);
29.     str += "CS 1704 is ";
30.     AddCheckStateToString(pCS1704, str);
31.     str += "CS 2704 is ";
32.     AddCheckStateToString(pCS2704, str);
33.     AfxMessageBox(str);
34. }

35. // This message map contains an entry to call a function
36. // when the Check States button is clicked.
37. BEGIN_MESSAGE_MAP(CCheckBoxDlg, CDialog)
38.     ON_COMMAND(IDC_CHECK_STATES, OnCheckStates)
39. END_MESSAGE_MAP()
```

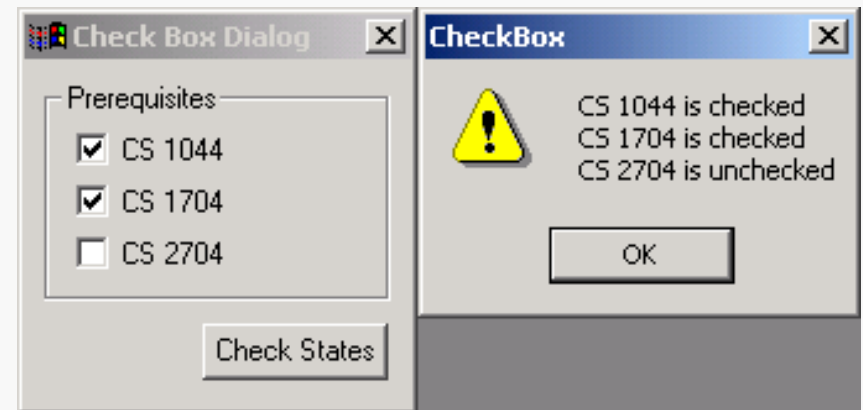
Use the custom `AddCheckStateToString()` function we wrote to quickly build an MFC `CString` that lists the states of the check boxes

When the “Check States” button is clicked, the `OnCheckStates()` function is called, which builds a string listing the states of each check box. Then `AfxMessageBox()` is called to display a message box with the generated string.

```
40. class CCheckBoxApp : public CWinApp
41. {
42. public:
43.     BOOL InitInstance()
44.     {
45.         // Create and show the dialog.
46.         CCheckBoxDlg dlg;
47.         dlg.DoModal();
48.         return FALSE;
49.     }
50. } checkBoxApp;
```

The *CheckBox* sample program will display the dialog to the right when run.

After checking or unchecking the controls, you can click the “Check States” button to display a message box that describes their states.



## Radio Buttons, Edit Boxes, List Boxes, and Combo Boxes

The next sample will introduce these last four control types, and put together everything you've seen so far in a more realistic application—a computerized order placement system for the imaginary fast food chain *MFC: Microsoft Fried Chicken*.

The next few slides give a quick introduction to each type of control, elaborates on some of the operations they perform, and where they are typically used.

Finally, the source code to the sample application is provided, with some extra information about functions that are used in the code on that slide.

Most comments have been removed from the source on the slides, to keep the length manageable. Also, error checking is minimal, again due to length—a real application would do much more verification and validation of the data being entered by the user.

## Radio Buttons

A radio button is similar to a check box, but only one of the buttons in a group can be selected at a given time. They are typically used to provide the user a choice among a small number of options.

The Windows radio button control, like the button and check box controls, is represented in MFC by the `CButton` class. In a Windows resource file, the `AUTORADIOBUTTON` statement is typically used to define them. (Like check boxes, a `RADIOBUTTON` statement is also provided that does not automatically change states when clicked, and is rarely used; unlike check boxes, there is no equivalent three-state control.)

To determine which control is checked, the `GetCheckedRadioButton()` function can be used to get the identifier of the checked button in a group, rather than querying the state of each button individually.



## Edit Boxes

An edit box is a control that allows the user to enter a numeric or text value. They can also be set to read-only, so that the user can see data specified in the control, but cannot edit it.

The difference between a read-only edit box and a static label is that the edit box can still be scrolled, and its contents can also be highlighted and copied to the clipboard.

Edit boxes are represented in MFC by the `CEdit` class, and in the resource file by the statement `EDITTEXT`. Many styles can be applied to these controls, and you can see a more in-depth discussion of them in most MFC/Windows reference books.

The `CEdit` class contains several member functions, but most often, you'll use the `GetDlgItemInt()` and `GetDlgItemText()` functions to retrieve values, and these are actually members of the parent window class, not the edit box itself.



## List Boxes

A list box is a control that displays a list of strings. They are typically used to provide feedback about a set of items for the user, or to allow selection from a large number of items.

The MFC class that encapsulates list boxes is `CListBox`; the corresponding resource statement is `LISTBOX`.

Unlike the previous types of controls, a list box is complex enough that the only way to manipulate it is to get a `CListBox*` pointer and call its member functions. There are many functions available, such as `AddString()`, `GetCount()`, `DeleteString()`, `ResetContent()`, and `GetCurSel()`, and these, among others, are illustrated in the *MFCChicken* sample application.

Items ordered:

Popcorn chicken (45 pieces) + Sauce	\$4.89
Regular, Extra Greasy	\$4.59
Hugel, Hot and Pricey	\$6.89
Small, Original Recipe	\$2.99

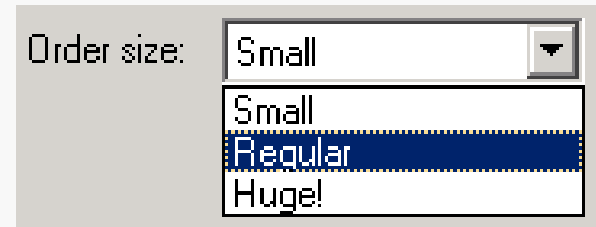
## Combo Boxes

A combo box is similar to a list box, except that the actual list is hidden until the drop-down button is clicked by the user.

Combo boxes can also optionally allow the selection area to be edited like an edit box—an example of this is the Address bar in Internet Explorer.

The MFC class that represents this control is `CComboBox`, and the resource statement is `COMBOBOX`. Since the list box and combo box share many similar operations, member functions like `AddString()` are mostly the same for both classes.

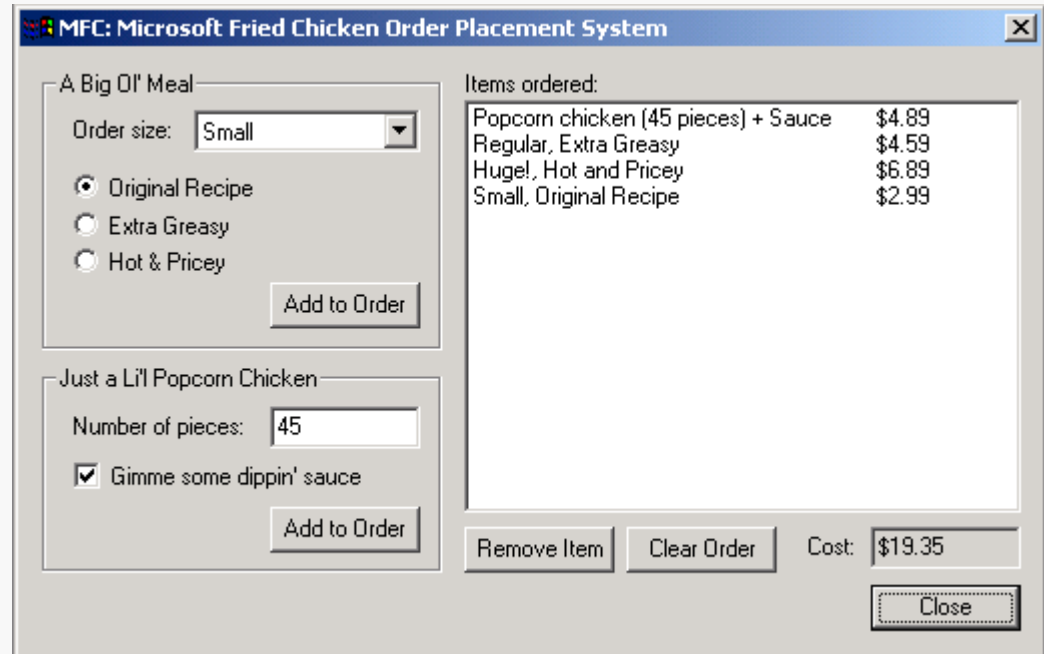
The height of the combo box control is fixed by the operating system—when you specify a height, you're actually setting the height of the drop-down list.



The sample application on the following slides has been written to illustrate the use of the seven control types you have seen so far, and how they interact to form a functional user-interface.

This sample is *not* designed to show you the most desirable GUI design for such an application—there are many changes that could be made to improve the program’s usability, and to tighten up the interface and the underlying code in general.

Any new functions or style flags that are introduced will be briefly described on that slide, under the code snippet where it was used. It would not be possible to explain every nuance of those functions in the space provided, so to find out more, check out the MSDN documentation that comes with Visual Studio.



```
1. // resource.h
2. //
3. #define IDD_MFCHICKENDLG          101
4. #define IDC_ORDERLIST            1000
5. #define IDC_ORDERSIZE            1001
6. #define IDC_RECIPE_ORIGINAL      1002
7. #define IDC_RECIPE_XGREASY       1003
8. #define IDC_RECIPE_HOTPRICEY     1004
9. #define IDC_ADDTOORDER_BIG       1005
10. #define IDC_ADDTOORDER_POPCORN   1006
11. #define IDC_DIPPING_SAUCE        1007
12. #define IDC_NUM_POPCORN          1008
13. #define IDC_REMOVEITEM           1009
14. #define IDC_CLEARORDER           1010
15. #define IDC_COST                  1011
```

Radio button resource identifiers should be sequentially numbered in a contiguous block

Notice that the resource identifiers for the recipe radio buttons (`IDC_RECIPE_ORIGINAL`, `IDC_RECIPE_XGREASY`, and `IDC_RECIPE_HOTPRICEY`) are sequentially numbered in a block (in fact, all the identifiers are, but the radio buttons are where it counts). This is necessary for the `GetCheckedRadioButton()` function to work later on in the code for the dialog box.

```
1. // MFChicken.rc
2. #include "resource.h"
3. #include "afxres.h"

4. IDD_MFCHICKENDLG DIALOG 0, 0, 340, 186
5. CAPTION "MFC: Microsoft Fried Chicken Order Placement System"
6. {
7.     COMBOBOX        IDC_ORDERSIZE, 58, 19, 75, 80,
8.                     CBS_DROPDOWNLIST | WS_VSCROLL | WS_TABSTOP
9.     AUTORADIOBUTTON "Original Recipe", IDC_RECIPE_ORIGINAL,
10.                    17, 38, 64, 10
11.     AUTORADIOBUTTON "Extra Greasy", IDC_RECIPE_XGREASY, 17, 50, 56, 10
12.     AUTORADIOBUTTON "Hot && Pricey", IDC_RECIPE_HOTPRICEY, 17, 61, 55, 10
13.     PUSHBUTTON      "Add to Order", IDC_ADDTOORDER_BIG, 83, 72, 50, 14
14.     EDITTEXT        IDC_NUM_POPCORN, 83, 110, 50, 13, ES_AUTOHSCROLL
15.     AUTOCHECKBOX    "Gimme some dippin' sauce", IDC_DIPPING_SAUCE,
16.                    18, 127, 99, 10
17.     PUSHBUTTON      "Add to Order", IDC_ADDTOORDER_POPCORN,
18.                    83, 141, 50, 14
```

The CBS\_DROPDOWNLIST combo box style lets the user choose an item from the list, but he or she cannot type into the edit portion.

Note the double ampersand (&&) in one of the captions above. Windows uses & as a prefix that underscores the following character, so && is needed to get a literal “&”.

```
19.     LISTBOX           IDC_ORDERLIST, 148, 16, 185, 127, LBS_USETABSTOPS |
20.                                     LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_TABSTOP
21.     PUSHBUTTON       "Remove Item", IDC_REMOVEITEM, 148, 147, 50, 14
22.     PUSHBUTTON       "Clear Order", IDC_CLEARORDER, 202, 147, 50, 14
23.     EDITTEXT          IDC_COST, 283, 147, 50, 13, ES_AUTOHSCROLL |
24.                                     ES_READONLY
25.     DEFPUSHBUTTON     "Close", IDOK, 283, 165, 50, 14
26.     LTEXT              "Items ordered:", IDC_STATIC, 148, 7, 46, 8
27.     GROUPBOX          "A Big Ol' Meal", IDC_STATIC, 7, 7, 133, 86
28.     LTEXT              "Order size:", IDC_STATIC, 17, 21, 34, 8
29.     GROUPBOX          "Just a Li'l Popcorn Chicken", IDC_STATIC,
30.                                     7, 97, 133, 64
31.     LTEXT              "Number of pieces:", IDC_STATIC, 17, 112, 58, 8
32.     LTEXT              "Cost:", IDC_STATIC, 262, 149, 17, 8
33. }
```

- The list box style `LBS_USETABSTOPS` should not be confused with the similarly named `WS_TABSTOP`. The first allows the items in the list box to be formatted using tabs, while the latter means that the control can be activated by cycling through the controls on the dialog with the `TAB` key.
- The `LBS_NOINTEGRALHEIGHT` style simply means that the height of the list box control will not be rounded down to the nearest multiple of the item height.
- Finally, `ES_READONLY` prevents the user from modifying the contents of the edit box.

```
1.  // MFChicken.h
2.  class CMFChickenDlg : public CDialog
3.  {
4.  public:
5.      CMFChickenDlg();
6.
7.      BOOL OnInitDialog();
8.
9.      // Message handlers for buttons
10.    	afx_msg void OnAddToOrderBig();
11.    	afx_msg void OnAddToOrderPopcorn();
12.    	afx_msg void OnRemoveItem();
13.    	afx_msg void OnClearOrder();
14.
15. private:
16.    	void UpdateCost();
17.
18.    	DECLARE_MESSAGE_MAP()
19. };
```

Override this virtual function in CDialog to perform more advanced initialization to the controls in the dialog

The only new item here is the declaration of the `OnInitDialog()` function, which is overriding a virtual function provided by the `CDialog` class to carry out certain initialization tasks that cannot be stored in the resource file.

```
19. // MFChicken.cpp
20. #include <afxwin.h>
21. #include "resource.h"
22. #include "MFChicken.h"

23. // Dialog constructor
24. CMFChickenDlg::CMFChickenDlg() :
25.     CDialog(IDD_MFCHICKENDLG) { }

26. // Connect the four main buttons in the dialog to
27. // message handler functions. The "Close" button
28. // is assigned the identifier IDOK, so it will be
29. // handled by default (closing the dialog box),
30. // so an entry for it does not need to be created.
31. BEGIN_MESSAGE_MAP(CMFChickenDlg, CDialog)
32.     ON_COMMAND(IDC_ADDTOORDER_BIG, OnAddToOrderBig)
33.     ON_COMMAND(IDC_ADDTOORDER_POPCORN, OnAddToOrderPopcorn)
34.     ON_COMMAND(IDC_REMOVEITEM, OnRemoveItem)
35.     ON_COMMAND(IDC_CLEARORDER, OnClearOrder)
36. END_MESSAGE_MAP()
```

The message map above again only contains `ON_COMMAND` event handlers to catch button presses on the dialog. A more complete application would want to monitor, for instance, when the selected item in the order list or the number of popcorn chicken pieces changes, and enable/disable buttons based on the new values.

```
37. const char* g_aSizeNames[] = {
38.     "Small", "Regular", "Huge!"
39. };

40. const char* g_aRecipeNames[] = {
41.     "Original Recipe", "Extra Greasy", "Hot and Pricey"
42. };

43. const double g_dPricePerPopcorn = 0.10;
44. const double g_dPriceOfSauce = 0.39;

45. const double g_aMealPrices[3][3] = {
46.     /*           Original      X-Greasy      Hot&Pricey */
47.     /* Small */      { 2.99,      3.79,      4.89      },
48.     /* Regular */    { 3.79,      4.59,      5.69      },
49.     /* Huge! */      { 4.99,      5.79,      6.89      }
50. };
```

The constants and tables above define values used later in the program, in order to keep all the item prices centralized and easily modifiable. The `g_aMealPrices` table is a 2-dimensional array that is indexed first by meal size, then by recipe.

```
54.  BOOL CMFChickenDlg::OnInitDialog() {
55.      CDialog::OnInitDialog();

56.      CComboBox* pOrderSize = (CComboBox*)GetDlgItem(IDC_ORDERSIZE);

57.      for(int i = 0; i < 3; i++)
58.          pOrderSize->AddString(g_aSizeNames[i]);

59.      pOrderSize->SetCurSel(1);

60.      CButton* pRecipeOriginal = (CButton*)GetDlgItem(IDC_RECIPE_ORIGINAL);
61.      pRecipeOriginal->SetCheck(BST_CHECKED);

62.      CListBox* pOrderList = (CListBox*)GetDlgItem(IDC_ORDERLIST);
63.      pOrderList->SetTabStops(135);
64.
65.      UpdateCost();

66.      return TRUE;
67. }
```

`CDialog::OnInitDialog()` is a virtual function that can be overridden to perform initialization that cannot be stored in the resource file. It will be invoked by the operating system after the dialog and its controls are created, but before it is displayed to the user.

```
69. void CMFChickenDlg::OnAddToOrderBig() {
70.     CComboBox* pOrderSize = (CComboBox*)GetDlgItem(IDC_ORDERSIZE);
71.     int nSize = pOrderSize->GetCurSel();
72.     int nRecipe = GetCheckedRadioButton(
73.         IDC_RECIPE_ORIGINAL, IDC_RECIPE_HOTPRICEY)
74.         - IDC_RECIPE_ORIGINAL;
75.     double dPrice = g_aMealPrices[nSize][nRecipe];
```

GetCheckedRadioButton uses the sequentially numbered identifiers in resource.h to determine which radio button in the group is checked.

Note how the program retrieves pointers to the controls it needs every time it enters the message handler, instead of asking for them during initialization and storing them. Why? Consider the `IDC_ORDERSIZE` combo box. During the execution of this program, no actual `CComboBox` object exists for this control. When `GetDlgItem()` is called, MFC allocates an object of type `CWnd`, associates the control with it, then returns a pointer to that object. These temporary objects are periodically cleaned up by MFC during the application's idle time, and thus are not guaranteed to exist outside the scope of the executing message handler.

The question then becomes: How is it possible to downcast from `CWnd*` to `CComboBox*`? If MFC allocates a `CWnd`, why can you treat it safely as a `CComboBox`? (If you want to be convinced that this works, see *Aside: Safe MFC Downcasts* at the end of this section.)

```
77.     CString strListItem;
78.     strListItem.Format("%s, %s\t$%3.2f",
79.         g_aSizeNames[nSize], g_aRecipeNames[nRecipe], dPrice);

80.     CListBox* pOrderList = (CListBox*)GetDlgItem(IDC_ORDERLIST);
81.     int nInsertedIdx = pOrderList->AddString(strListItem);
82.     pOrderList->SetItemData(nInsertedIdx, (DWORD)(dPrice * 100));

83.     UpdateCost();
84. }
```

The `CString` class is an MFC-based string class similar to the one in the standard C++ libraries, and it is used throughout the MFC class hierarchy. The `Format()` member function performs `printf`-style formatting on a variable number of parameters—see the documentation for `printf` in any textbook or documentation that covers C-style I/O for an explanation of the formatting codes.

The `CListBox::AddString()` function adds an item to the end of the list and returns the actual index where it was placed.

Each item in a list box or combo box has an associated 32-bit value that can be used to store some extra customized data. In this case, the price of the item is stored here, first multiplied by 100 to remove the fractional part and cast to a `DWORD` (because a `double` is 64-bits; a `DWORD` is defined to be an unsigned `int`, which is 32-bits.)

```
85. void CMFChickenDlg::OnAddToOrderPopcorn() {
86.     int nNumPieces = GetDlgItemInt(IDC_NUM_POPCORN, NULL, FALSE);

87.     if(1 <= nNumPieces && nNumPieces <= 100) {
88.         double dPrice = nNumPieces * g_dPricePerPopcorn;

89.         CString strListItem;

90.         CButton* pDippingSauce =
91.             (CButton*)GetDlgItem(IDC_DIPPING_SAUCE);

92.         if(pDippingSauce->GetCheck() == BST_CHECKED) {
93.             dPrice += g_dPriceOfSauce;

94.             strListItem.Format("Popcorn chicken (%d pieces) + "
95.                 "Sauce\t$%3.2f", nNumPieces, dPrice);
96.         }
97.         else {
98.             strListItem.Format("Popcorn chicken (%d pieces)\t$%3.2f",
99.                 nNumPieces, dPrice);
100.        }
```

The `GetDlgItemInt()` function is a `CWnd` member function that queries the contents of the edit box associated with the specified resource ID and returns the result as an integer.

```
105.         CListBox* pOrderList = (CListBox*)GetDlgItem(IDC_ORDERLIST);
106.         int nInsertedIdx = pOrderList->AddString(strListItem);
107.         pOrderList->SetItemData(nInsertedIdx, (DWORD)(dPrice * 100));

108.         UpdateCost();
109.     }
110.     else {
111.         AfxMessageBox("Please enter a number between 1 and 100, "
112.             "inclusive.");
113.     }
114. }
```

This message handler is very similar to the previous one (`OnAddToOrderBig()`), but it also verifies that the value entered by the user in the “Number of pieces” field is valid (in this case, between 1 and 100, inclusive). If it is not, `AfxMessageBox()` is called to warn the user.

(This is less than desirable error handling, however, and is only used to keep this example simple. A more complete application would instead monitor the control’s contents by trapping `EN_CHANGE` notifications from the edit box using a message handler, and disable the “Add to Order” button when an invalid value is entered.)

```
115. void CMFChickenDlg::OnRemoveItem() {
116.     CListBox* pOrderList = (CListBox*)GetDlgItem(IDC_ORDERLIST);
117.     int nSelectedItem = pOrderList->GetCurSel();

118.     if(nSelectedItem != LB_ERR) {
119.         pOrderList->DeleteString(nSelectedItem);
120.         UpdateCost();
121.     }
122.     else {
123.         AfxMessageBox("Please select an item to remove from the order.");
124.     }
125. }

126. void CMFChickenDlg::OnClearOrder() {
127.     CListBox* pOrderList = (CListBox*)GetDlgItem(IDC_ORDERLIST);
128.     pOrderList->ResetContent();

129.     UpdateCost();
130. }
```

`CListBox::GetCurSel()` returns the index of the selected item in the list box, or the predefined constant `LB_ERR` if there is no item selected.

`CListBox::DeleteString()` removes the item at the specified index from the list, and `CListBox::ResetContent()` removes all the items from the list.

```
131. void CMFChickenDlg::UpdateCost() {
132.     CListBox* pOrderList = (CListBox*)GetDlgItem(IDC_ORDERLIST);

133.     int nOrderCount = pOrderList->GetCount();
134.     double dCost = 0.0;

135.     for(int i = 0; i < nOrderCount; i++) {
136.         DWORD dwItemData = pOrderList->GetItemData(i);
137.         double dItemPrice = (double)dwItemData / 100.0;

138.         dCost += dItemPrice;
139.     }

140.     CString strCost;
141.     strCost.Format("$%3.2f", dCost);

142.     SetDlgItemText(IDC_COST, strCost);
143. }
```

The `UpdateCost()` function calls `CListBox::GetCount()` to retrieve the number of items in the list, then iterates through each one and sums the price information that is stored in each item's 32-bit user data value.

`SetDlgItemText()` is then called to set the contents of the "Cost" edit box to a string that contains the dollar amount of the entire order.

```
144. class CMFChickenApp : CWinApp
145. {
146. public:
147.     BOOL InitInstance()
148.     {
149.         // Create and show the dialog.
150.         CMFChickenDlg dlg;
151.         dlg.DoModal();
152.         return FALSE;
153.     }
154. } chickenApp;
```

This completes the *MFChicken* sample application. If you want to see it in action, you can download the source code from the course website and compile it on your machine.

Try putting some breakpoints in the message handlers and stepping through them to understand the values that MFC is returning to the application (but make sure the Visual Studio window isn't maximized and you can see both windows on the screen without having to hide one or the other—you won't be able to bring the *MFChicken* dialog to the foreground while the debugger is active inside a message handler).

Why can a pointer to a `CWnd` object be safely downcast to `CComboBox*`, or to any other derived control type?

In your OOP lecture, you were probably told that downcasts are usually unsafe. Imagine that you have a class `BaseClass`, and another class `DerivedClass` that inherits `BaseClass`. If you allocate an object of type `BaseClass`, you cannot necessarily treat it as type `DerivedClass` if `DerivedClass` contains additional data, because that data wasn't created when the `BaseClass` object was allocated. Any attempt to access such data would result in an access violation (or at the very least, logical errors), because the data just doesn't exist.

MFC takes advantage of technicalities in the design and implementation of C++ to make this downcast possible. If you take a look at the MFC source code, you'll notice that none of the basic control classes (`CButton`, `CEdit`, `CComboBox`, etc.) have any data members—they only provide methods that act on data in the parent class, `CWnd`. Therefore, when you cast from a base class to a derived class, there is no risk that you will attempt to access data that doesn't exist.

Ok, the derived classes don't have any data of their own. But I'm still not convinced that the method calls to the derived class will still work.

To understand this, you must consider the implementation of a class in C++, and most object-oriented languages in general. Recall that only a single copy of any given method exists for all the objects of an entire class. When this method is called, the object pointer is passed as an implicit parameter to the function, so the code knows which object to operate on. (You should recognize this pointer as the keyword `this` in C++.)

Therefore, calling a `CComboBox` member function on a `CWnd` pointer is harmless—the implicit `this` pointer that gets passed into the `CComboBox` function will point to the `CWnd` object that MFC allocated earlier. Since the method only accesses `CWnd` data members, the call will succeed, because what MFC allocated *was* a `CWnd` object.

To convince you of this, the next slide shows an example of an MFC control class member function.

Below is the definition of the `CComboBox::GetCurSel()` function (from `afxwin2.inl`), stripped of extraneous MFC preprocessor definitions and formatted for legibility:

```
int CComboBox::GetCurSel() const {  
    ASSERT(::IsWindow(m_hWnd));  
    return (int)::SendMessage(m_hWnd, CB_GETCURSEL, 0, 0);  
}
```

The only data member accessed by this function is `m_hWnd`, which is a member of `CWnd` and represents a unique Windows handle value that identifies the control.

`::IsWindow()` is a global function defined by Windows that verifies that this handle is valid (points to an existing control).

`::SendMessage()` is another global Windows function that performs one of a number of operations on the control handle specified. In this case, `CB_GETCURSEL` is a preprocessor-`#define`'d integer that asks the control to return the index of the currently selected item.

Hopefully now you're convinced that while this may not be the purest OO design, it does in fact produce the correct results through some clever C++ tricks.