Module 5: The Modal Dialog and Windows Common Controls

Program examples compiled using Visual C++ 6.0 (MFC 6.0) compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this Tutorial are listed below:

The Modal Dialog and Windows Common Controls

Almost every Windows-based program uses a dialog window to interact with the user. The dialog might be a simple **OK** message box, or it might be a complex data entry form. Calling this powerful element a dialog "box" is an injustice. A dialog is truly a window that receives messages, that can be moved and closed, and that can even accept drawing instructions in its client area. The two kinds of dialogs are **modal** and **modeless**.

Modal vs. Modeless Dialogs

The `CDialog` base class supports both modal and modeless dialogs. With a **modal dialog**, such as the **Open File** dialog, the user cannot work elsewhere in the same application more correctly, in the same user interface thread until the dialog is closed. With a **modeless dialog**, the user can work in another window in the application while the dialog remains on the screen. 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. Your choice of a modal or a modeless dialog depends on the application. Modal dialogs are much easier to program, which might influence your decision.

Resources and Controls

So now you know a dialog is just a window. What makes the dialog different from the `CView` windows you've seen already? For one thing, a dialog window is almost always tied to a Windows resource that identifies the dialog's elements and specifies their layout. Because you can use the dialog editor that is one of the resource editors to create and edit a dialog resource, you can quickly and efficiently produce dialogs in a visual manner.
A dialog contains a number of elements called **controls**. Dialog controls include edit controls (aka text boxes), buttons, list boxes, combo boxes, static text (aka labels), tree views, progress indicators, sliders, and so forth. Windows manages these controls using special **grouping** and **tabbing** logic and that relieves you of a major programming burden. The dialog controls can be referenced either by a `CWnd` pointer (because they are really windows) or by an index number (with an associated `#define` constant) assigned in the resource. A control sends a message to its parent dialog in response to a user action such as typing text or clicking a button.
The MFC Library and ClassWizard work together to enhance the dialog logic that Windows provides. ClassWizard generates a class derived from `CDialog` and then lets you associate dialog class **data members** with dialog controls. You can specify editing parameters such as maximum text length and numeric high and low limits. ClassWizard generates statements that call the MFC **data exchange** and **data validation** functions to move information back and forth between the screen and the data members.

Programming a Modal Dialog

Modal dialogs are the most frequently used dialogs. A user action (a menu choice, for example) brings up a dialog on the screen, the user enters data in the dialog, and then the user closes the dialog. Here's a summary of the steps to add a modal dialog to an existing project:

1. Use the dialog editor to create a dialog resource that contains various controls. 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.
2. Use ClassWizard to create a dialog class that is derived from `CDialog` and attached to the resource created in step 1. ClassWizard adds the associated code and header file to the Microsoft Visual C++ project.

   When ClassWizard generates your derived dialog class, it generates a constructor that invokes a `CDialog` modal constructor, which takes a resource ID as a parameter. Your generated dialog header file contains a class enumerator constant `IDD` that is set to the dialog resource ID. In the CPP file, the constructor implementation looks something 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 file.
3. Use ClassWizard to add data members, exchange functions, and validation functions to the dialog class.
4. Use ClassWizard to add message handlers for the dialog's buttons and other event-generating controls.
5. 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). The `OnOK()` is called by default.
6. 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.

Now we'll proceed with a real example, one step at a time.

The MYMFC7 Example

Let's not mess around with wimpy little dialogs. We'll build a monster dialog that contains almost every kind of control. The job will be easy because Visual C++'s dialog editor is there to help us. The finished product is shown in the following Figure.

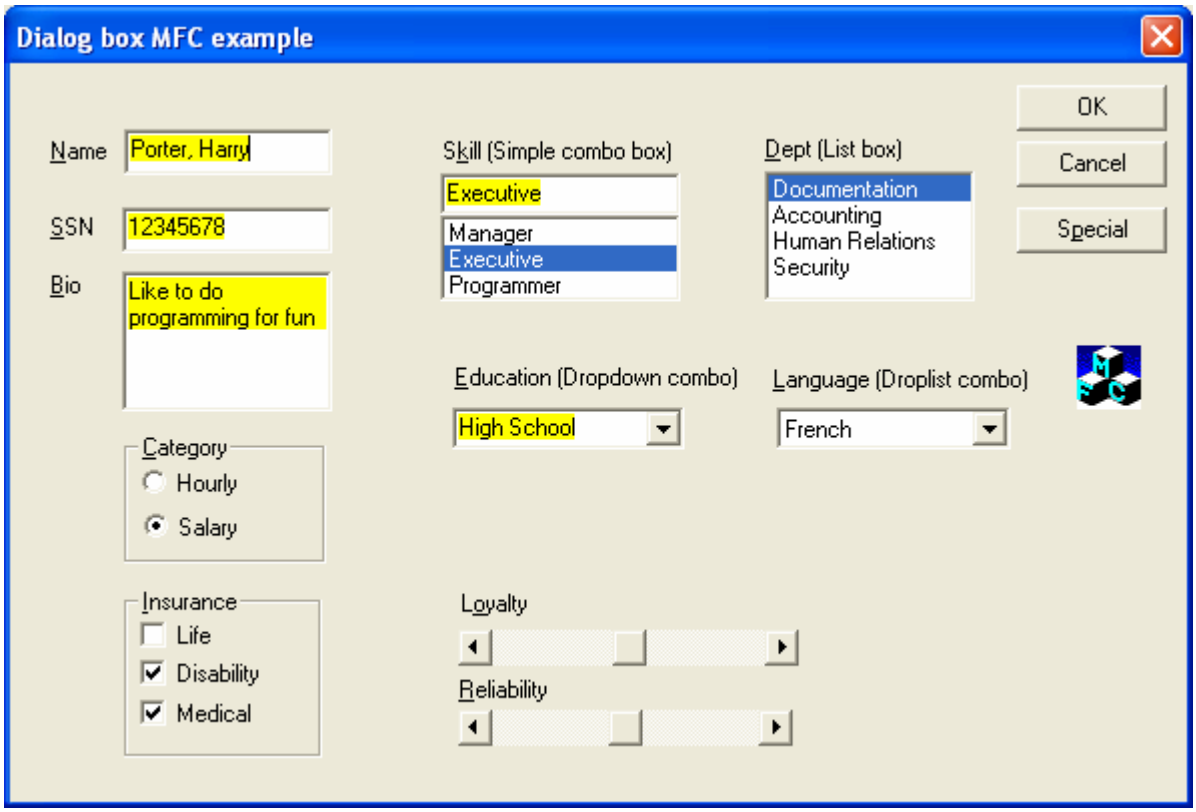


Figure 1: The finished dialog of the MYMFC7 project.

As you can see, the dialog supports a human resources application. These kinds of business programs are fairly boring, so the challenge is to produce something that could not have been done with 80-column punched cards. The program is brightened a little by the use of scroll bar controls for "**Loyalty**" and "**Reliability**." Here is a classic example of direct action and visual representation of data! Later on, ActiveX controls could add more interest.

Building the Dialog Resource

Here are the steps for building the dialog resource:

Run AppWizard to generate a project called MYMFC7. Choose **New** from **Visual C++'s File** menu, and then click the **Projects** tab and select **MFC AppWizard (exe)**. Accept all the defaults but two: select **Single Document** and deselect **Printing And Print Preview**. The options and the default class names are shown here.

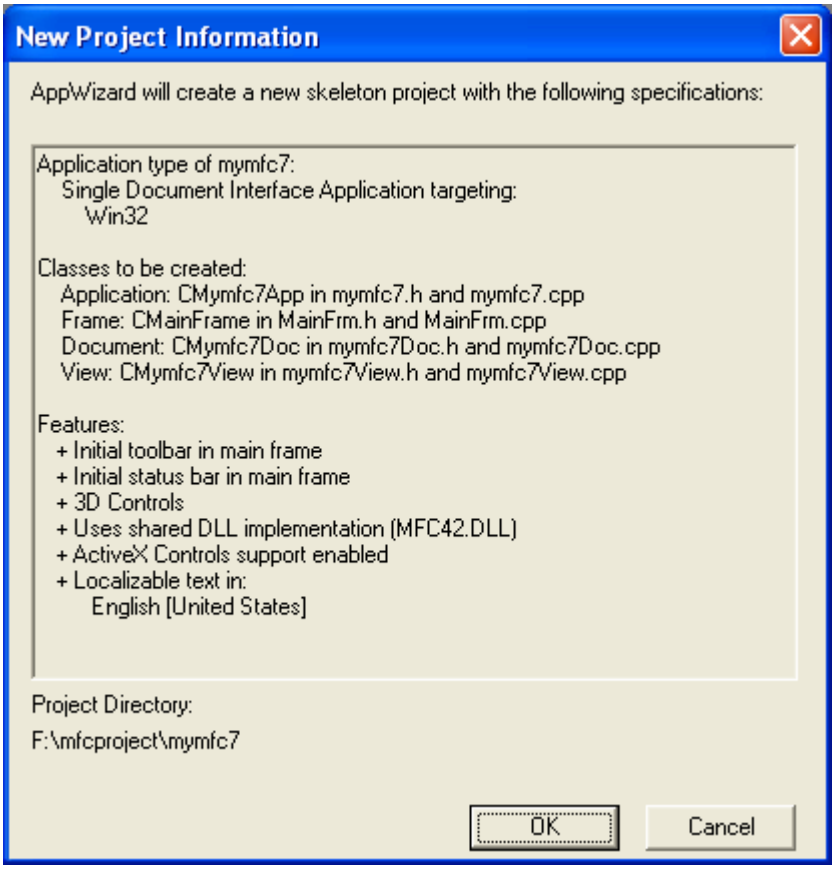


Figure 2: MYMFC7 project summary.

As usual, AppWizard sets the new project as the current project.

Create a new dialog resource with ID `IDD_DIALOG1` (the default given ID). Choose **Resource** from Visual C++'s **Insert** menu. The **Insert Resource** dialog appears. Click on **Dialog**, and then click **New**. Visual C++ creates a new dialog resource, as shown here.

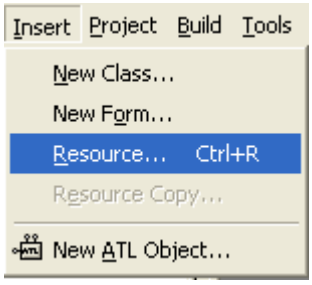


Figure 3: Creating a new dialog resource through the **Insert** menu.

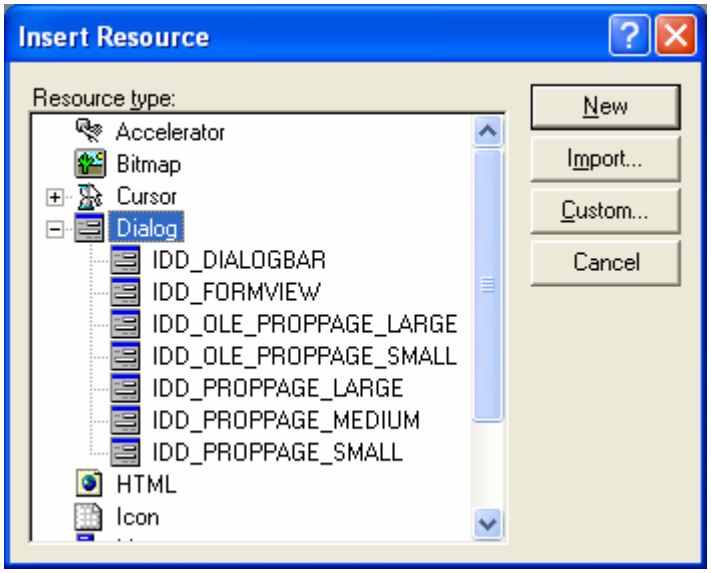


Figure 4: Inserting new dialog resource.

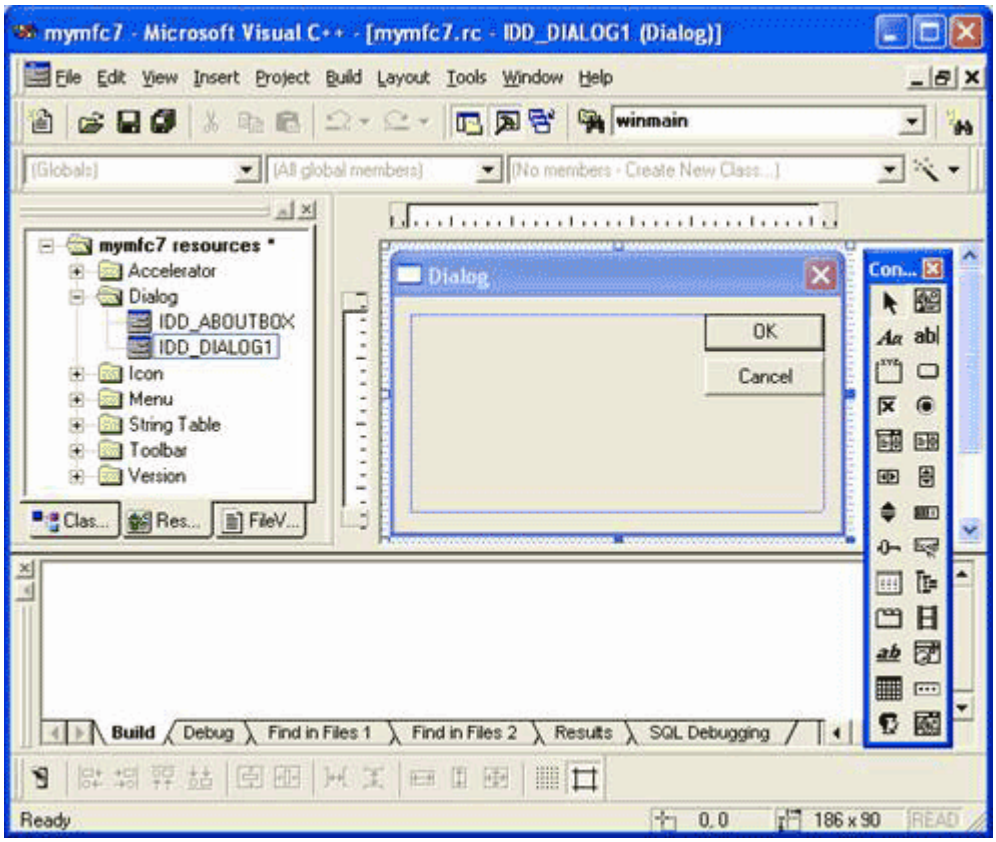


Figure 5: Dialog resource editor.

The dialog editor assigns the resource `ID IDD_DIALOG1` to the new dialog. Notice that the dialog editor inserts **OK** and **Cancel** buttons for the new dialog.

Resize the dialog and assign a `MyDialog` caption. Enlarge the dialog box to about 5-by-7 inches.
When you right-click on the new dialog and choose **Properties** from the pop-up menu, the **Dialog Properties** dialog appears. Type in the caption for the new dialog as shown in the screen below.

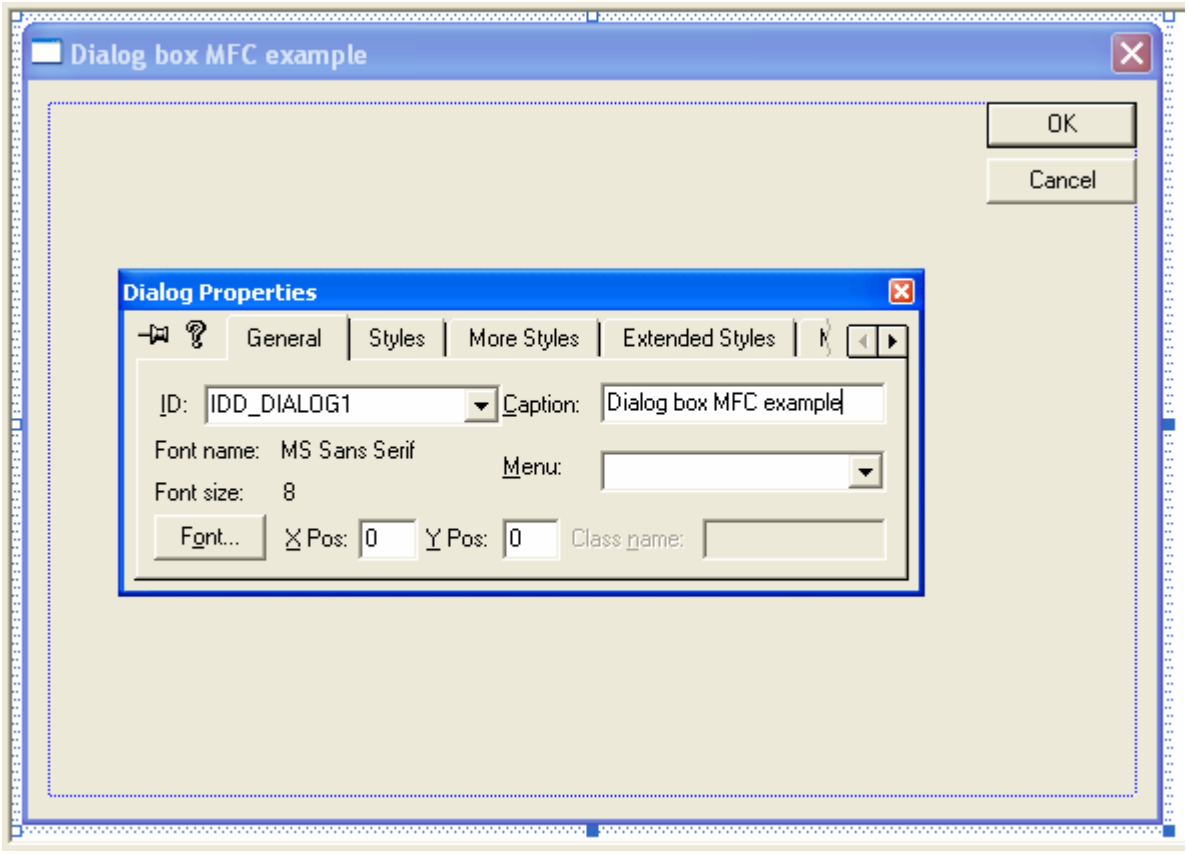


Figure 6: Dialog's properties.

The state of the **pushpin button** in the upper-left corner determines whether the **Dialog Properties** dialog stays on top of other windows. When the pushpin is "pushed," the dialog stays on top of other windows.

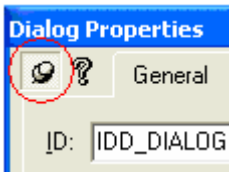


Figure 7: The pushpin button, making the **Dialog Properties** staying on top.

Click the **Toggle Grid** button (on the **Dialog** toolbar) to reveal the grid and to help align controls.

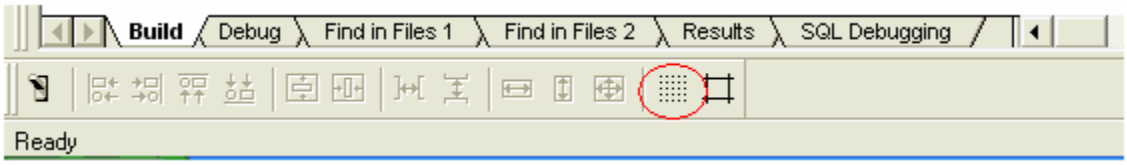


Figure 8: The dialog grid, helping the controls alignment on the dialog.

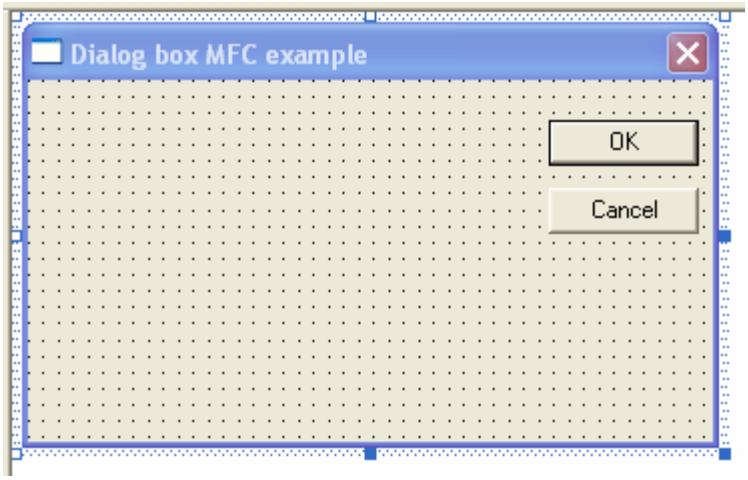


Figure 9: Dialog with grid.

You can test your dialog during the design stage by clicking the **Test** switch in the **Dialog toolbar**.

Figure 10: The dialog **Test** switch, testing your dialog during the design process.

Similar buttons and other utilities can also be accessed through the **Layout** menu.

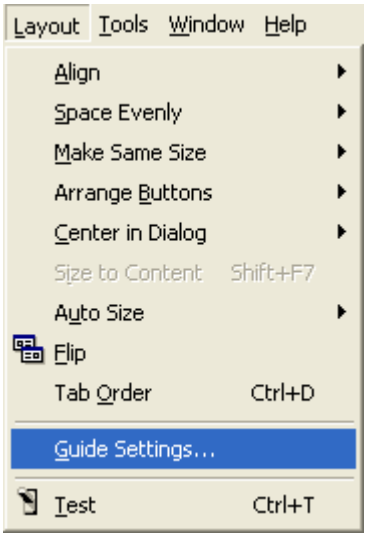


Figure 11: Accessing dialog's editing utilities through Layout menu.

Set the dialog style. Click on the **Styles** tab at the top of the **Dialog Properties** dialog, and then set the style properties as shown in the following illustration.

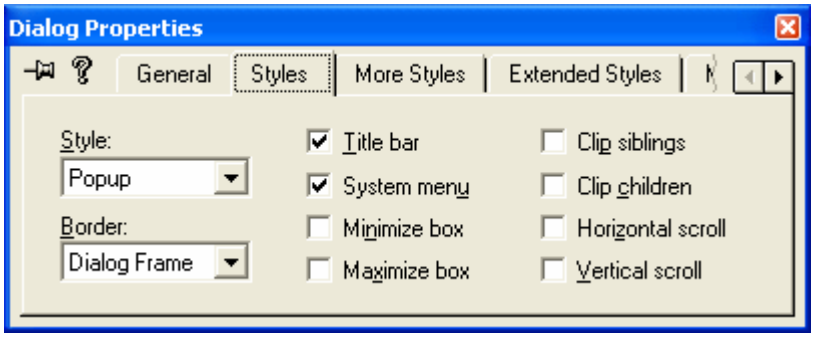


Figure 12: Setting the dialog style.

Set additional dialog styles. Click on the **More Styles** tab at the top of the **Dialog Properties** dialog, and then set the style properties as shown here.

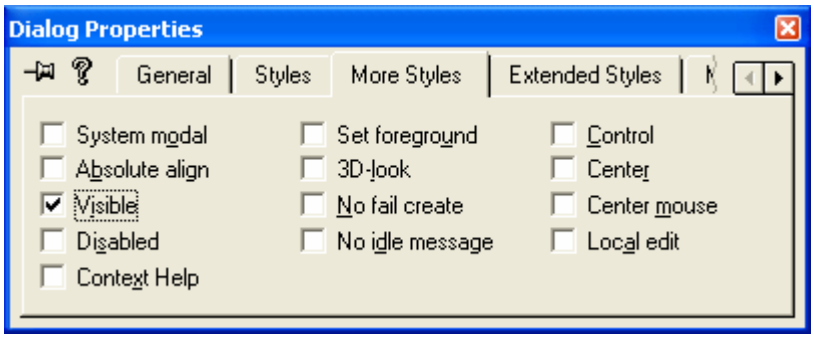
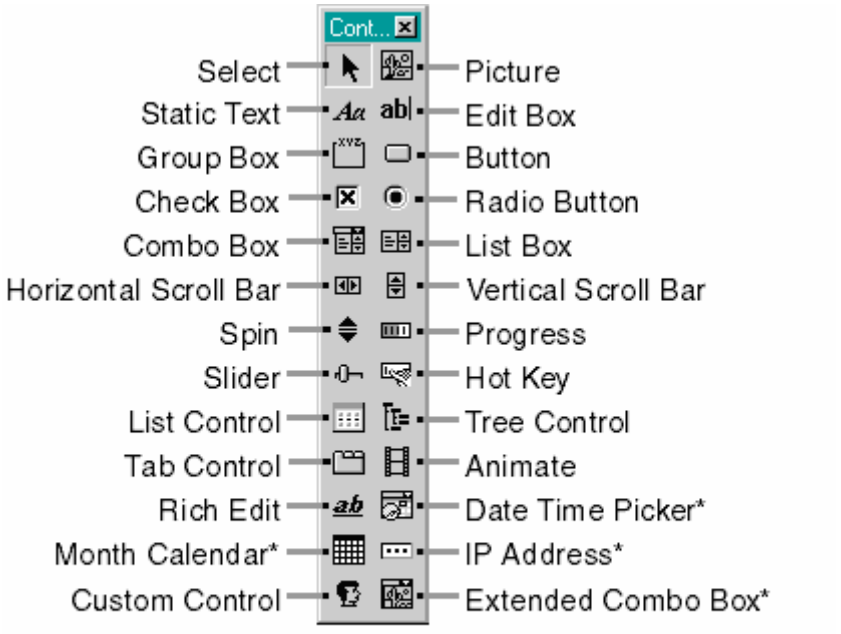


Figure 13: Setting additional dialog styles.

Add the dialog's controls. Use the control palette to add each control. If the control palette is not visible, right-click any toolbar and choose **Controls** from the list. Drag controls from the **control palette** (shown below) to the new dialog and then position and size the controls, as shown in Figure 1. Here are the control palette's controls.



*Indicates a new Internet Explorer 4 common control introduced in Visual C++ 6.0.

Figure 14: Available controls from Visual C++ control palette.

The dialog editor displays the position and size of each control in the status bar. The position units are special "dialog units," or **DLU**s, not device units. A horizontal DLU is the average width of the dialog font divided by 4. A vertical DLU is the average height of the font divided by 8. The dialog font is normally 8-point MS Sans Serif.

Here's a brief description of the dialog's controls, use drag and drop for the controls:

- The **static text** control for the **Name** field. A static text control simply paints characters on the screen. No user interaction occurs at runtime. You can type the text after you position the bounding rectangle, and you can resize the rectangle as needed. This is the only static text control you'll see listed in text, but you should also create the other static text controls as shown earlier in Figure 6-1. Follow the same procedure for the other static text controls in the dialog. All static text controls have the same ID, but that doesn't matter because the program doesn't need to access any of them.

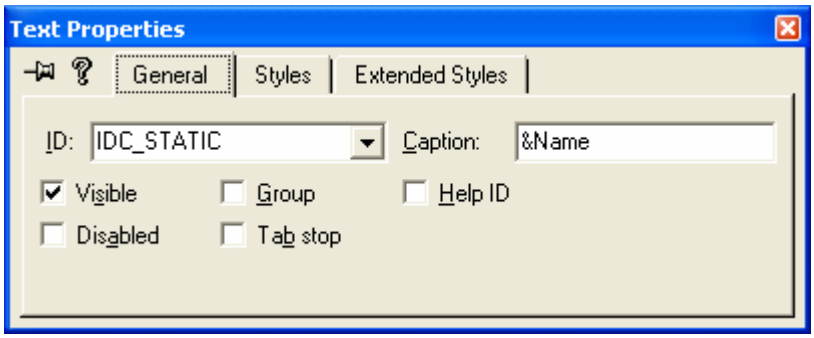


Figure 15: Setting The **static text** control.

Keyboard Accelerator

A static text control (such as **Name** or **Skill**) has an ampersand (**&**) embedded in the text for its caption. At runtime, the ampersand will appear as an underscore under the character that follows. This keyboard accelerator or short cut key, enables the user to jump to selected controls by holding down the **Alt** key (or **Ctrl** or **Shift**) and pressing the key corresponding to the underlined character. The related control must immediately follow the static text in the tabbing order. Thus, **`Alt-N`** jumps to the Name edit control and **`Alt-K`** jumps to the Skill combo box. Needless to say, designated jump characters should be unique within the dialog. The Skill control uses Alt-K because the SSN control uses **`Alt-S`**. Unfortunately the keyboard accelerator not works in this example because we need extra step to make it function and will be shown in another Module. The extra step is setting the key through the Accelerator resource as shown below.

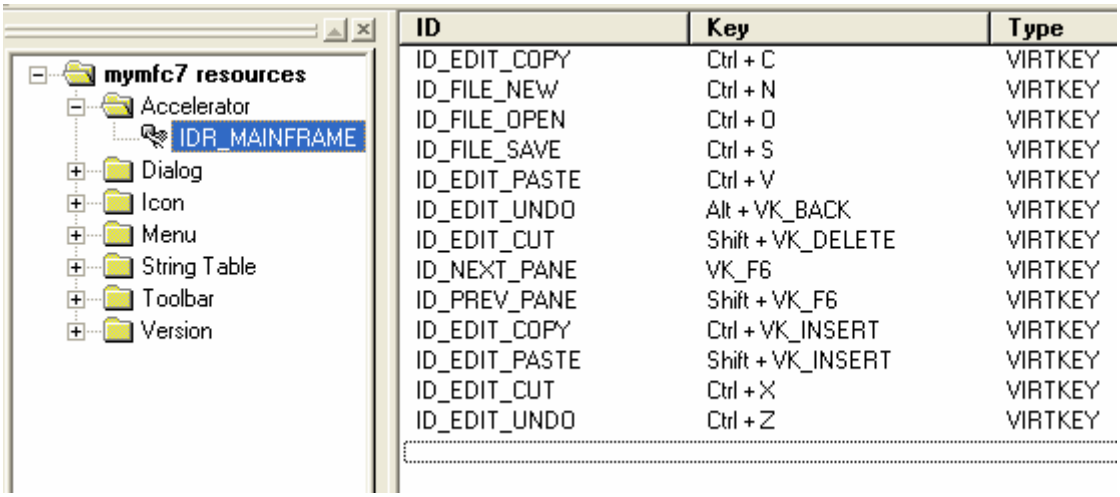


Figure 16: Assigning the keyboard accelerator through the ResourceView.

- The **Name** edit control. An **edit** control is the primary means of entering text in a dialog. Right-click the control, and then choose **Properties**. Change this control's ID from `IDC_EDIT1` to `IDC_NAME`. Accept the defaults for the rest of the properties. Notice that the default sets `Auto HScroll`, which means that the text scrolls horizontally when the box is filled.
- The **SSN** (social security number) edit control. As far as the dialog editor is concerned, the SSN control is exactly the same as the Name edit control. Simply change its ID to `IDC_SSN`. Later you will use ClassWizard to make this a numeric field.

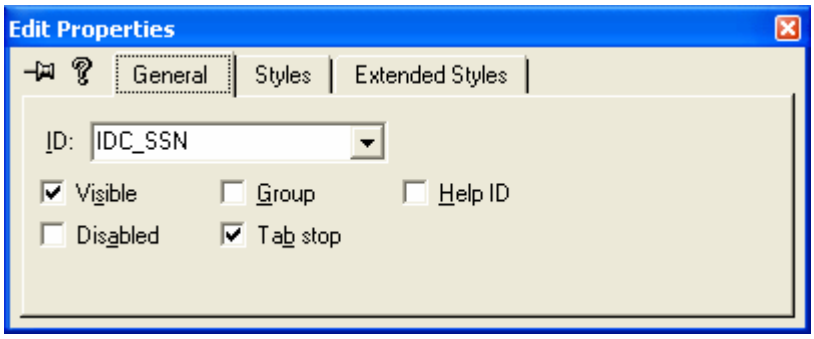


Figure 17: Modifying the **SSN** edit control properties.

- The **Bio** (biography) edit control. This is a multiline edit control. Change its ID to `IDC_BIO`, and then set its properties as shown here.

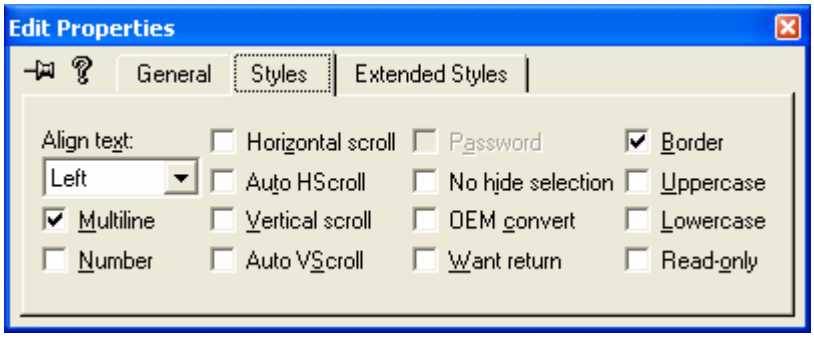


Figure 18: Modifying the **Bio** (biography) edit control properties.

- The **Category** group box. This control serves only to group two radio buttons visually. Type in the caption **Category**. The default ID is sufficient.
- The **Hourly** and **Salary** radio buttons. Position these radio buttons inside the **Category** group box. Set the **Hourly** button's ID to `IDC_CAT` and set the other properties as shown here.

Figure 19: Modifying the radio button properties.

Figure 20: Modifying the second radio button properties.

Be sure that both buttons have the **Auto** property (the default) on the **Styles** tab set and that only the **Hourly** button has the **Group** property set. When these properties are set correctly, Windows ensures that only one of the two buttons can be selected at a time. The **Category** group box has no effect on the buttons' operation.

- The **Insurance** group box. This control holds three check boxes. Type in the caption **Insurance**. Later, when you set the dialog's tab order, you'll ensure that the **Insurance** group box follows the last radio button of the **Category** group. Set the **Insurance** control's **Group** property now in order to "terminate" the previous group. If you fail to do this, it isn't a serious problem, but you'll get several warning messages when you run the program through the debugger.

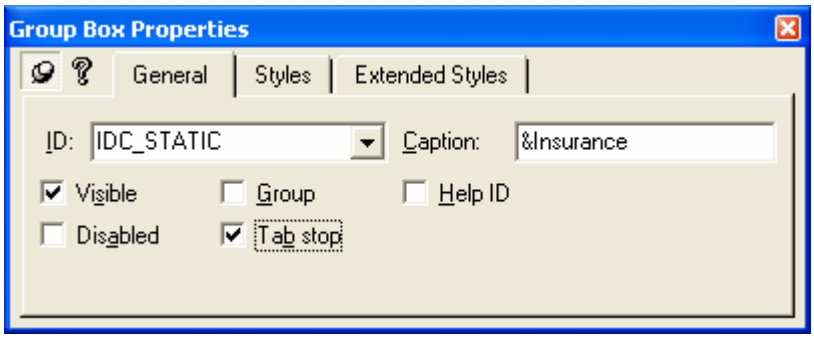


Figure 21: Modifying the **Insurance** group box properties.

- The **Life**, **Disability**, and **Medical** check boxes. Place these controls inside the Insurance group box. Accept the default properties, but change the IDs to `IDC_LIFE`, `IDC_DIS`, and `IDC_MED`. Unlike radio buttons, check boxes are independent; the user can set any combination.

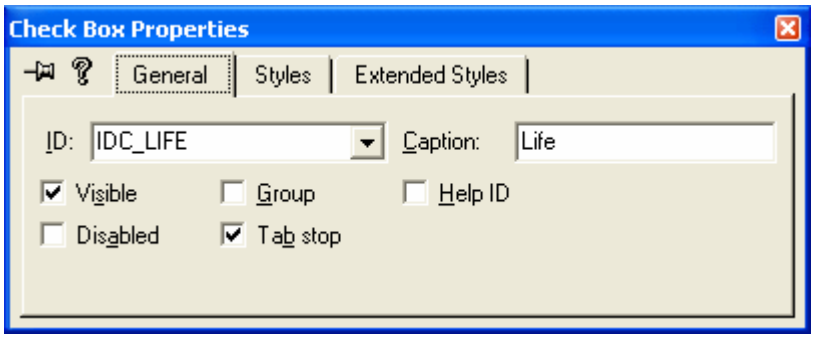


Figure 22: Modifying the **Life**, **Disability**, and **Medical** check boxes properties.

- The **Skill** combo box. This is the first of three types of combo boxes. Change the ID to `IDC_SKILL`, and then click on the **Styles** tab and set the **Type** option to **Simple**. Click on the **Data** tab, and add three skills (terminating each line with **Ctrl-Enter**) in the **Enter Listbox Items** box.

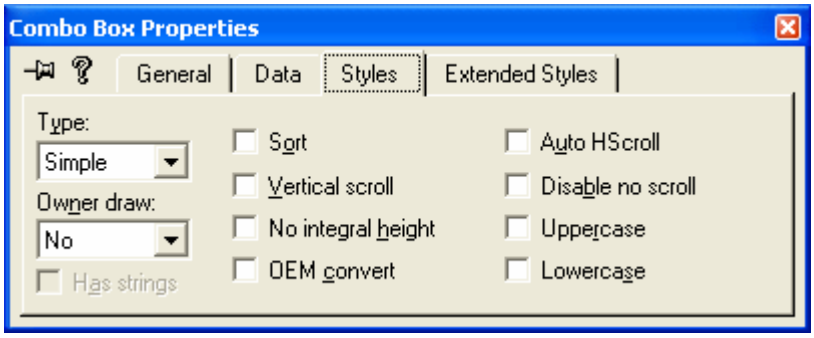


Figure 23: Modifying the **Skill** combo box properties.

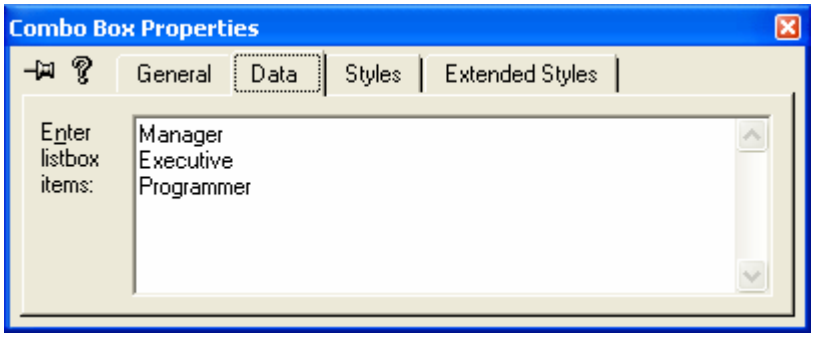


Figure 24: Adding the listbox item.

This is a combo box of type **Simple**. The user can type anything in the top edit control; use the mouse to select an item from the attached list box, or use the **Up** or **Down** direction key to select an item from the attached list box.

- The **Educ** (education) combo box. Change the ID to `IDC_EDUC`; otherwise, accept the defaults. Add the three education levels in the **Data** page, as shown in Figure 25. In this Dropdown combo box, the user can type anything in the edit box, click on the arrow, and then select an item from the drop-down list box or use the **Up** or **Down** direction key to select an item from the attached list box.

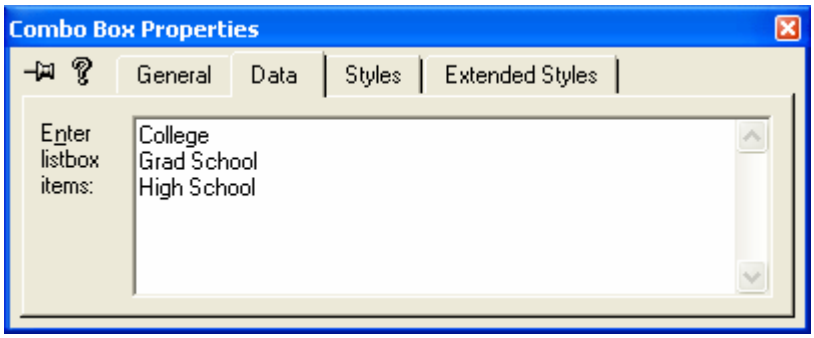


Figure 25: Modifying the **Educ** (education) combo box.

Aligning Controls

To align two or more controls, select the controls by clicking on the first control and then Shift-clicking on the other controls you want to align. Next choose one of the alignment commands (Left, Horiz.Center, Right, Top, Vert.Center, or Bottom) from the **Align** submenu on the dialog editor's **Layout** menu. To set the size for the drop-down portion of a combo box, click on the box's arrow and drag down from the center of the bottom of the rectangle.

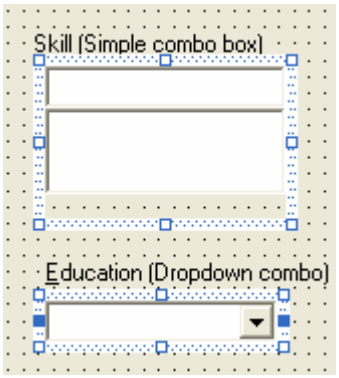


Figure 26: Aligning controls on the dialog.

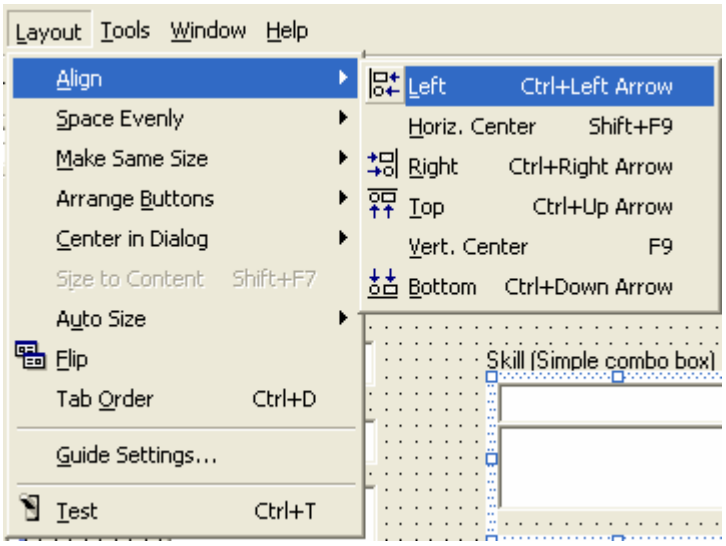


Figure 27: Using sub menus under the **Layout** menu for controls alignment etc.

- The **Dept** (department) list box. Change the ID to `IDC_DEPT`; otherwise, accept all the defaults. In this list box, the user can select only a single item by using the mouse, by using the **Up** or **Down** direction key, or by typing the first character of a selection. Note that you can't enter the initial choices in the dialog editor. You'll see how to set these choices later.

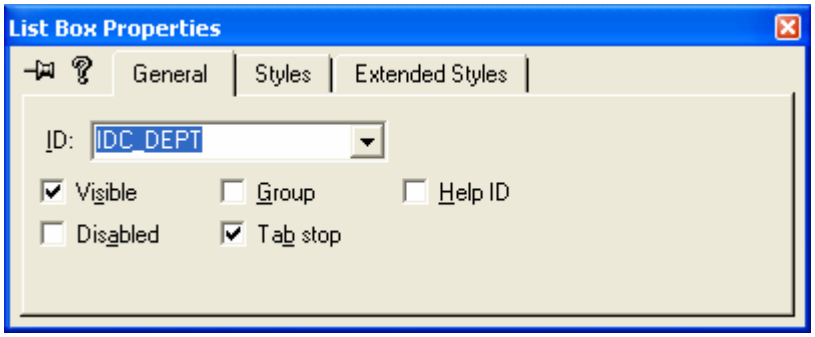


Figure 28: Modifying the **Dept** (department) list box properties.

- The **Lang** (language) combo box. Change the ID to `IDC_LANG`, and then click on the **Styles** tab and set the **Type** option to `Drop List`. Add three languages (English, French, and Spanish) in the **Data** page. With this **Drop List** combo box, the user can select only from the attached list box. To select, the user can click on the

arrow and then select an entry from the drop-down list or the user can type in the first letter of the selection and then refine the selection using the **Up** or **Down** direction key.

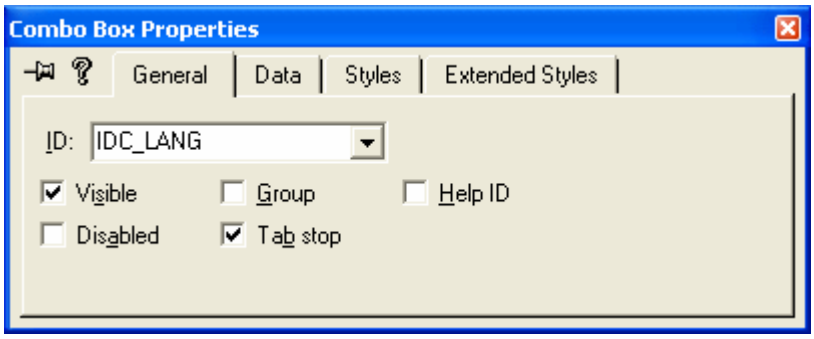


Figure 29: Modifying the **Lang** (language) combo box.

- The **Loyalty** and **Reliability** scroll bars. Do not confuse scroll bar controls with a window's built-in scroll bars as seen in scrolling views. A scroll bar control behaves in the same manner as do other controls and can be resized at design time. Position and size the horizontal scroll bar controls as shown previously in Figure 1, and then assign the IDs `IDC_LOYAL` and `IDC_RELY`.

Figure 30: Modifying the **Loyalty** and **Reliability** scroll bar properties.

Selecting a Group of Controls

To quickly select a group of controls, position the mouse cursor above and to the left of the group. Hold down the left mouse button and drag to a point below and to the right of the group, as shown here.

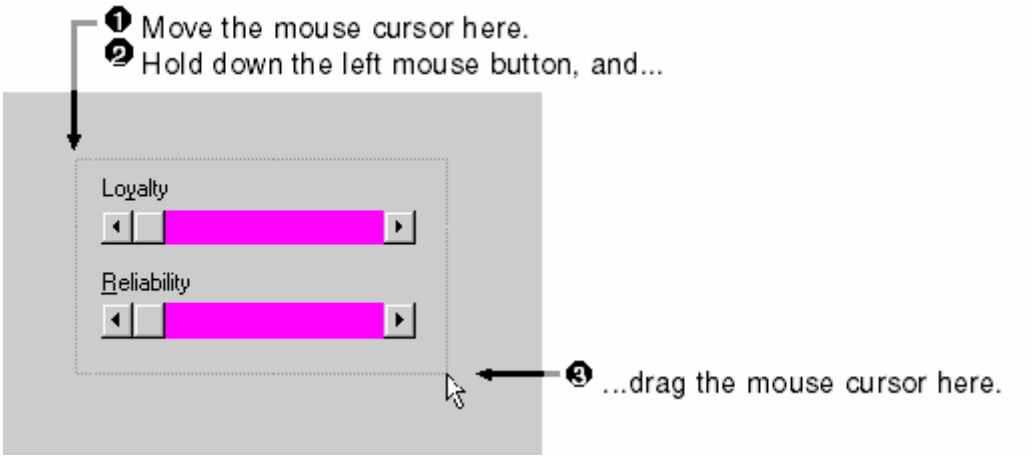


Figure 31: How to select a group of controls on the dialog.

- The **OK**, **Cancel**, and **Special** pushbuttons. Be sure the button captions are **OK**, **Cancel**, and **Special**, and then assign the ID `IDC_SPECIAL` to the **Special** button. Later you'll learn about special meanings that are associated with the default IDs `IDOK` and `IDCANCEL`.

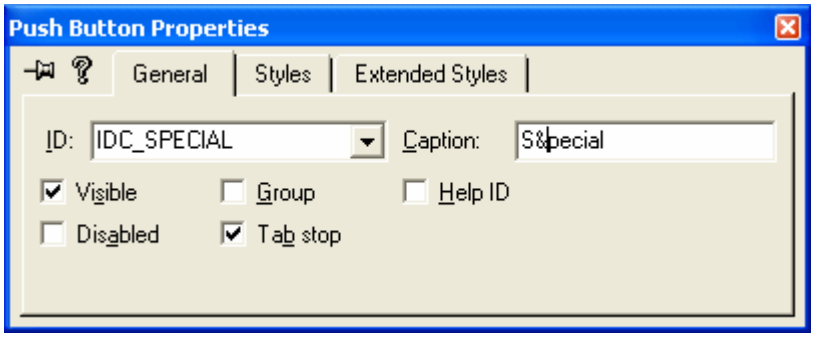


Figure 32: Modifying the **Special** pushbutton properties.

- Any icon. (The MFC icon is used as an example.) You can use the **Picture** control to display any icon or bitmap in a dialog, as long as the icon or bitmap is defined in the resource script. We'll use the program's MFC icon, identified as `IDR_MAINFRAME`. Set the **Type** option to **Icon**, and set the **Image** option to `IDR_MAINFRAME`. Leave the ID as `IDC_STATIC`.

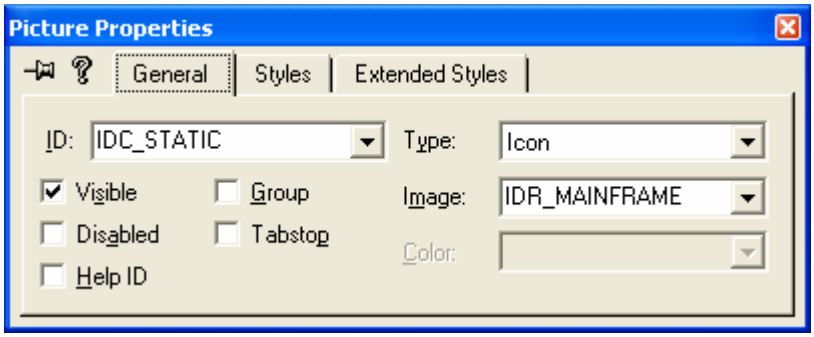


Figure 33: Using the **Picture** control to display any icon.

Check the dialog's tabbing order. Choose **Tab Order** from the dialog editor's **Layout** menu. Use the mouse to set the tabbing order shown below. Click on each control in the order shown, and then press **Enter**.

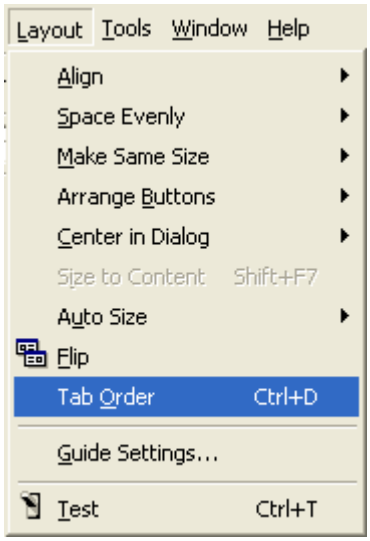


Figure 34: Viewing and setting the tab order.

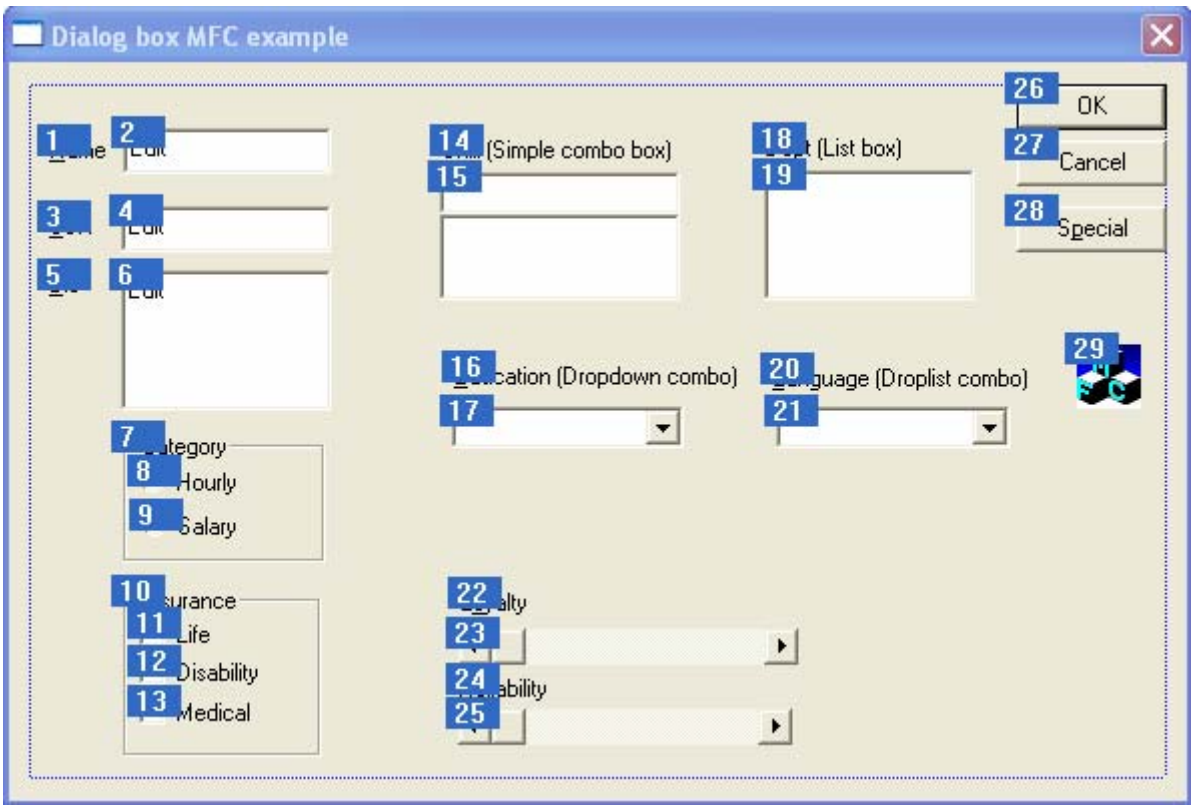


Figure 35: The tab order of the controls.

If you mess up the tab sequence partway through, you can recover with a Ctrl-left mouse click on the last correctly sequenced control. Subsequent mouse clicks will start with the next sequence number.

Save the resource file on disk. For safety, choose **Save** from the **File** menu or click the **Save** button on the toolbar to save **mymfc7.rc**. Keep the dialog editor running, and keep the newly built dialog on the screen.

ClassWizard and the Dialog Class

You have now built a dialog resource, but you can't use it without a corresponding dialog class. (The section titled "Understanding the MYMFC7 Application" explains the relationship between the dialog window and the underlying classes.) ClassWizard works in conjunction with the dialog editor to create that class as follows:

Choose ClassWizard from Visual C++'s **View** menu (or press Ctrl-W). Be sure that you still have the newly built dialog, `IDD_DIALOG1`, selected in the dialog editor and that MYMFC7 is the current Visual C++ project.

Add the `CMymfc7Dialog` class. ClassWizard detects the fact that you've just created a dialog resource without an associated C++ class. It politely asks whether you want to create a class, as shown below.

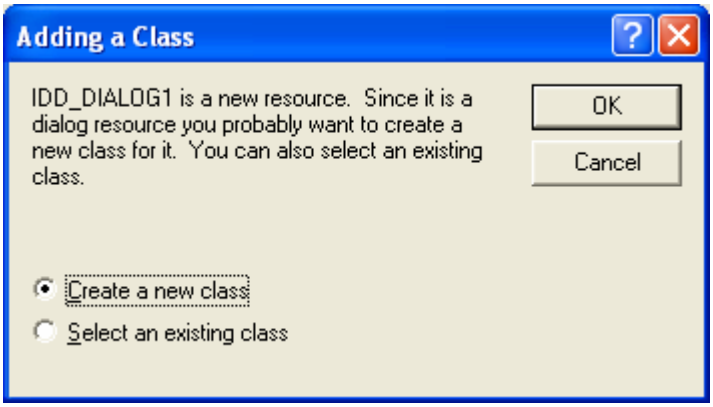


Figure 36: A new class creation dialog prompt.

Accept the default selection of **Create A New Class**, and click **OK**. Fill in the top field of the **New Class** dialog, as shown here.

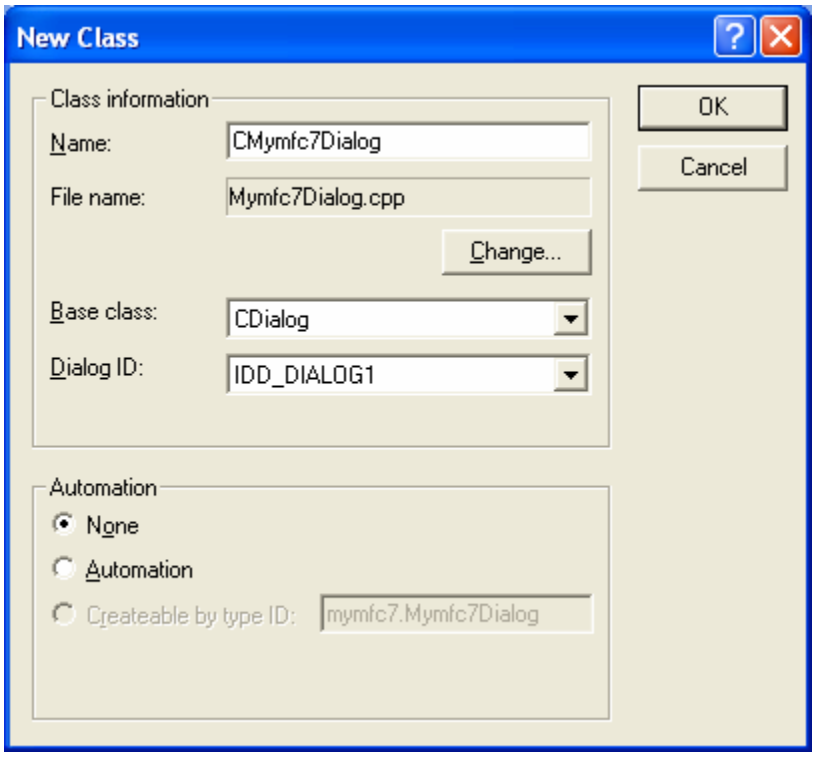


Figure 37: New class information dialog.

Add the `CMymfc7Dialog` variables. After ClassWizard creates the `CMymfc7Dialog` class, the MFC ClassWizard dialog appears. Click on the **Member Variables** tab, and the **Member Variables** page appears, as shown here.

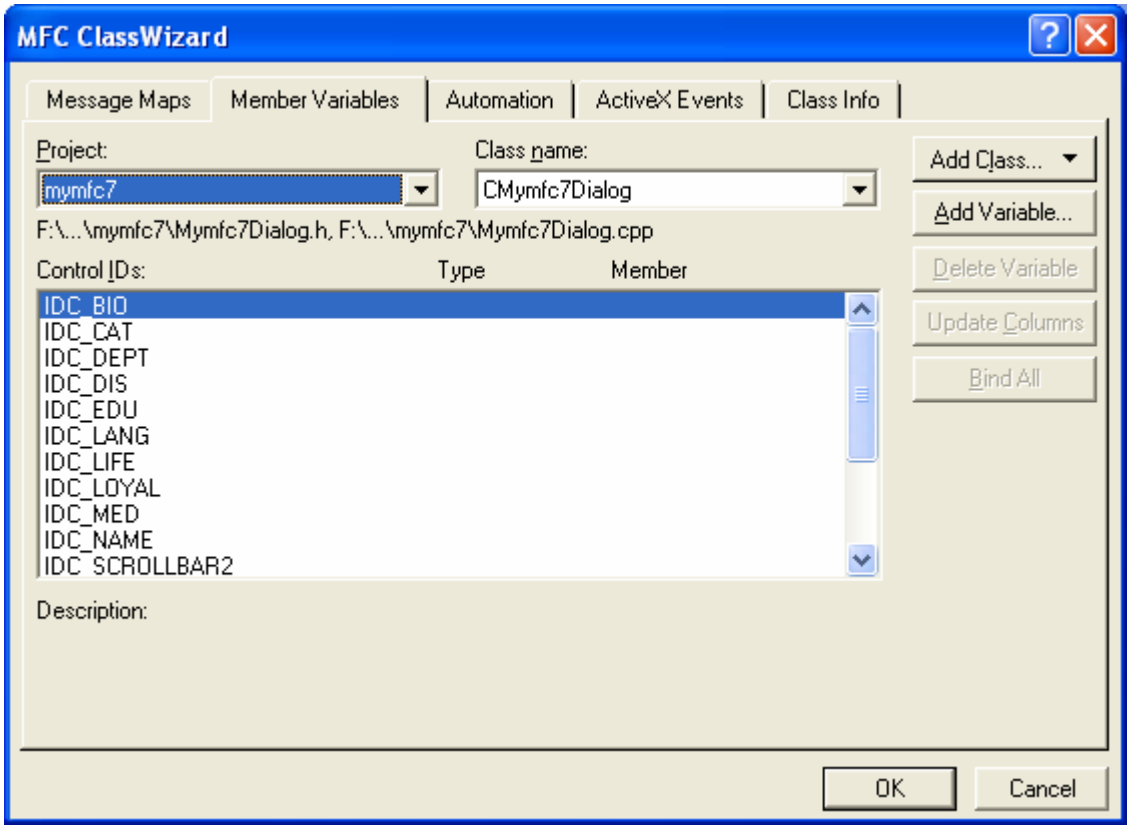


Figure 38: Adding the `CMymfc7Dialog` variables through the **Member Variables** page of the ClassWizard.

You need to associate data members with each of the dialog's controls. To do this, click on a control ID and then click the **Add Variable** button. The **Add Member Variable** dialog appears, as shown in the following illustration.

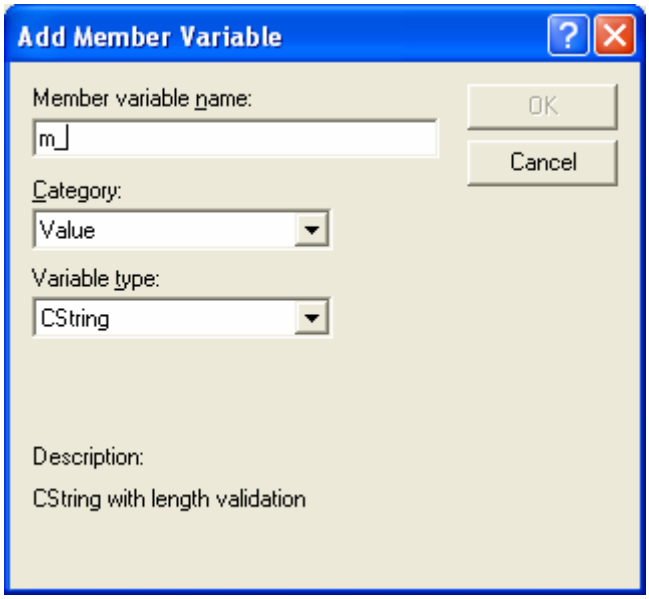


Figure 39: Adding member variable's category and type.

Type in the member variable name, and choose the variable type according to the following table. Be sure to type in the member variable name exactly as shown; the case of each letter is important. For the **Category**, select `Value` for all the

member variable. When you're done, click **OK** to return to the MFC ClassWizard dialog. Repeat this process for each of the listed controls.

Control ID	Data Member	Type
IDC_BIO	m_strBio	CString
IDC_CAT	m_nCat	int
IDC_DEPT	m_strDept	CString
IDC_DIS	m_bInsDis	BOOL
IDC_EDUC	m_strEduc	CString
IDC_LANG	m_nLang	int
IDC_LIFE	m_bInsLife	BOOL
IDC_LOYAL	m_nLoyal	int
IDC_MED	m_bInsMed	BOOL
IDC_NAME	m_strName	CString
IDC_RELY	m_nRely	int
IDC_SKILL	m_strSkill	CString
IDC_SSN	m_nSsn	int

Table 1: Member variables for MYMFC7 controls

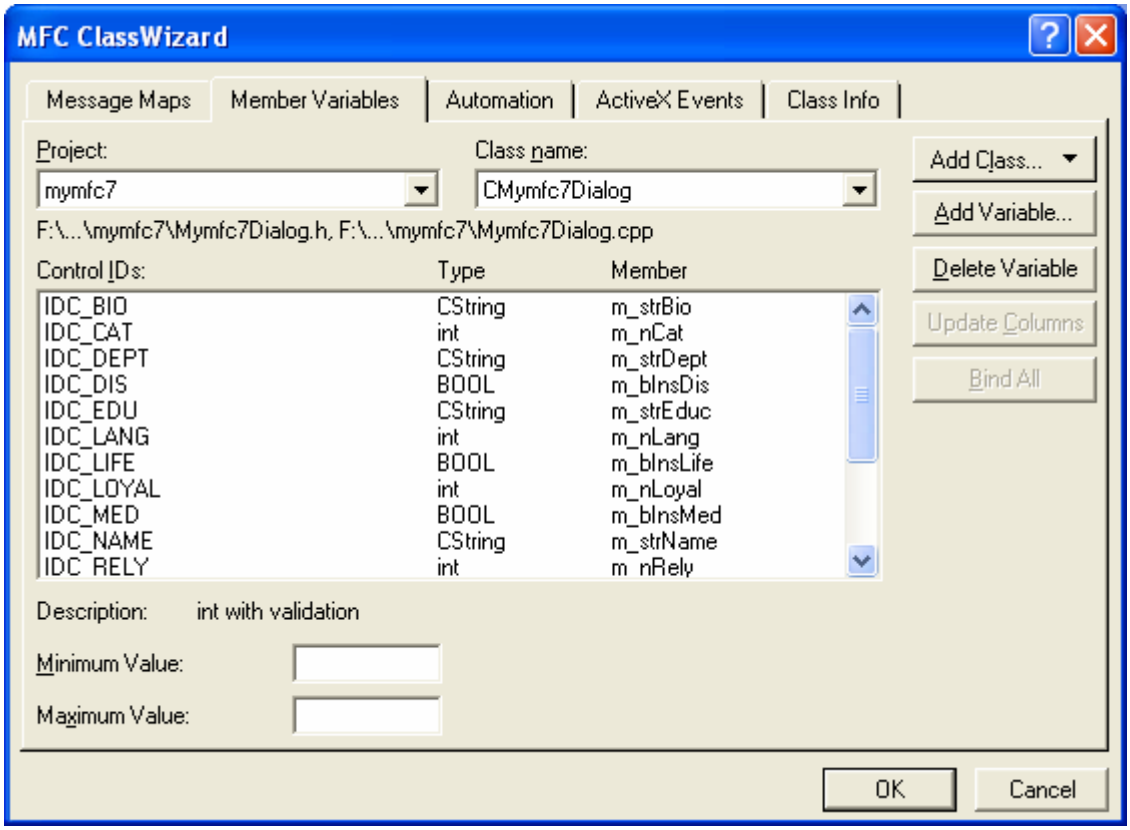


Figure 40: Member variables added when seen through the ClassWizard's **Member Variables** page.

As you select controls in the MFC ClassWizard dialog, various edit boxes appear at the bottom of the dialog. If you select a `CString` variable, you can set its maximum number of characters; if you select a numeric variable, you can set its high and low limits. Set the minimum value for `IDC_SSN` to 0 and the maximum value to 999999999.

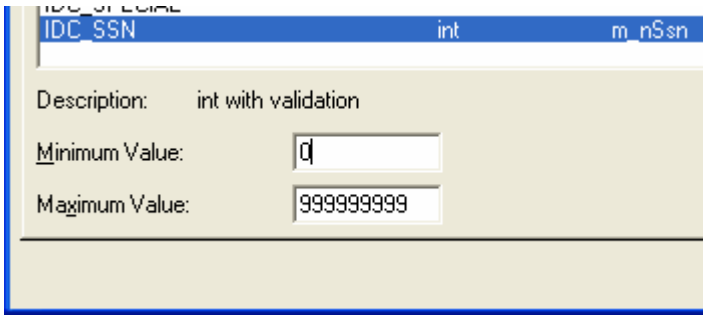


Figure 41: Setting the minimum and maximum value of the member variable for data validation.

Most relationships between control types and variable types are obvious. The way in which radio buttons correspond to variables is not so intuitive, however. The `CDialog` class associates an integer variable with each radio button group, with the first button corresponding to value 0, the second to 1, and so forth.

Add the message-handling function for the **Special** button. `CMymfc7Dialog` doesn't need many message-handling functions because the `CDialog` base class, with the help of Windows, does most of the dialog management. When you specify the ID `IDOK` for the **OK** button (ClassWizard's default), for example, the virtual `CDialog` function `OnOK()` gets called when the user clicks the button. For other buttons, however, you need message handlers.
Click on the **Message Maps** tab. The ClassWizard dialog should contain an entry for `IDC_SPECIAL` in the **Object IDs** list box. Click on this entry, and double-click on the `BN_CLICKED` message that appears in the **Messages** list box. ClassWizard invents a member function name, `OnSpecial()`, and opens the **Add Member Function** dialog, as shown here.

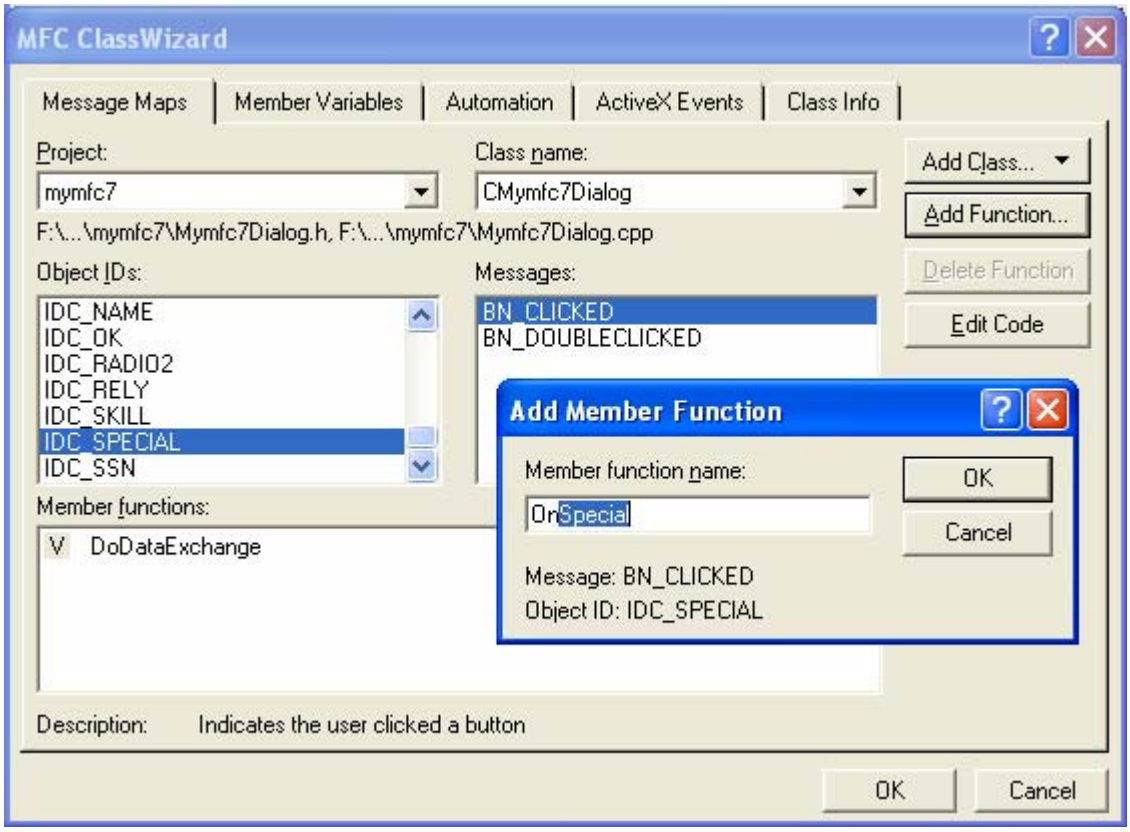


Figure 42: Adding the message-handling function for the **Special** button.

You could type in your own function name here, but this time accept the default and click **OK**. Click the **Edit Code** button in the MFC ClassWizard dialog. This opens the file **mymfc7Dialog.cpp** and moves to the `OnSpecial()`

function. Insert a TRACE statement in the OnSpecial() function by typing in the code shown below, which replaces the existing code:

```
void CMymfc7Dialog::OnSpecial()
{
    TRACE("CMymfc7Dialog::OnSpecial\n");
}
```

```
void CMymfc7Dialog::OnSpecial()
{
    // TODO: Add your control notification handler code here
    TRACE("CMymfc7Dialog::OnSpecial\n");

}
```

Listing 1.

Use ClassWizard to add an OnInitDialog() message-handling function. As you'll see in a moment, ClassWizard generates code that initializes a dialog's controls. This DDX (Dialog Data Exchange) code won't initialize the list-box choices, however, so you must override the CDialog::OnInitDialog function. Although OnInitDialog() is a virtual member function, ClassWizard generates the prototype and skeleton if you map the WM_INITDIALOG message in the derived dialog class.
To do so, click on CMymfc7Dialog in the **Object IDs** list box and then double-click on the WM_INITDIALOG message in the Messages list box. Click the **Edit Code** button in the MFC ClassWizard dialog to edit the OnInitDialog() function.

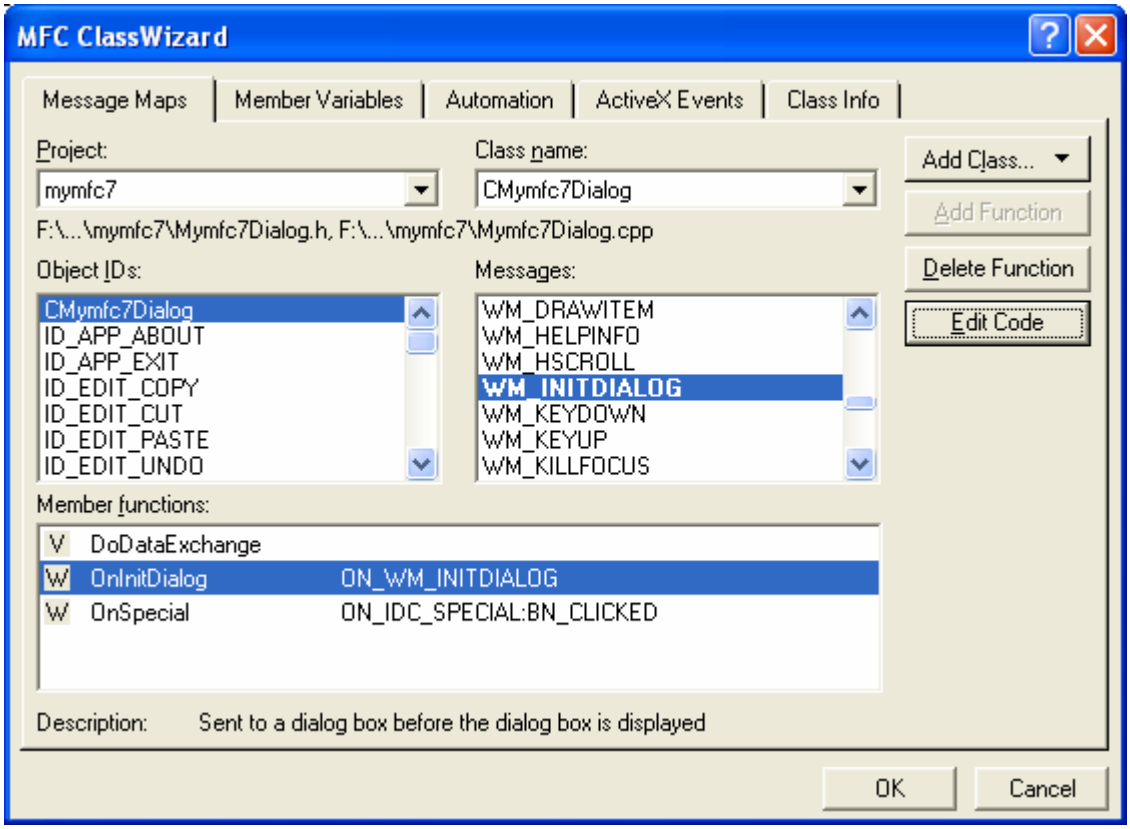


Figure 43: Adding an OnInitDialog() message-handling function through the ClassWizard.

Type in the following code, which replaces the existing code:

```
BOOL CMymfc7Dialog::OnInitDialog()
```

```
    {
        // Be careful to call CDialog::OnInitDialog
        // only once in this function
        CListBox* pLB = (CListBox*) GetDlgItem(IDC_DEPT);
        pLB->InsertString(-1, "Documentation");
        pLB->InsertString(-1, "Accounting");
        pLB->InsertString(-1, "Human Relations");
        pLB->InsertString(-1, "Security");

        // Call after initialization
        return CDialog::OnInitDialog();
    }
```

```
BOOL CMymfc7Dialog::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: Add extra initialization here
    // Be careful to call CDialog::OnInitDialog
    // only once in this function
    CListBox* pLB = (CListBox*) GetDlgItem(IDC_DEPT);
    pLB->InsertString(-1, "Documentation");
    pLB->InsertString(-1, "Accounting");
    pLB->InsertString(-1, "Human Relations");
    pLB->InsertString(-1, "Security");

    // Call after initialization
    return CDialog::OnInitDialog();
}
```

Listing 2.

You could also use the same initialization technique for the combo boxes, in place of the initialization in the resource.

Connecting the Dialog to the View

Now we've got the resource and the code for a dialog, but it's **not connected to the view**. In most applications, you would probably use a menu choice to activate a dialog, but we haven't studied menus yet. Here we'll use the familiar mouse-click message `WM_LBUTTONDOWN` to start the dialog. The steps are as follows:

1. In ClassWizard, select the `CMymfc7View` class. At this point, be sure that MYMFC7 is Visual C++'s current project.
2. Use ClassWizard to add the `OnLButtonDown()` member function. You've done this in the examples in earlier Modules. Simply select the `CMymfc7View` class name, click on the `CMymfc7View` object ID, and then double-click on `WM_LBUTTONDOWN`.

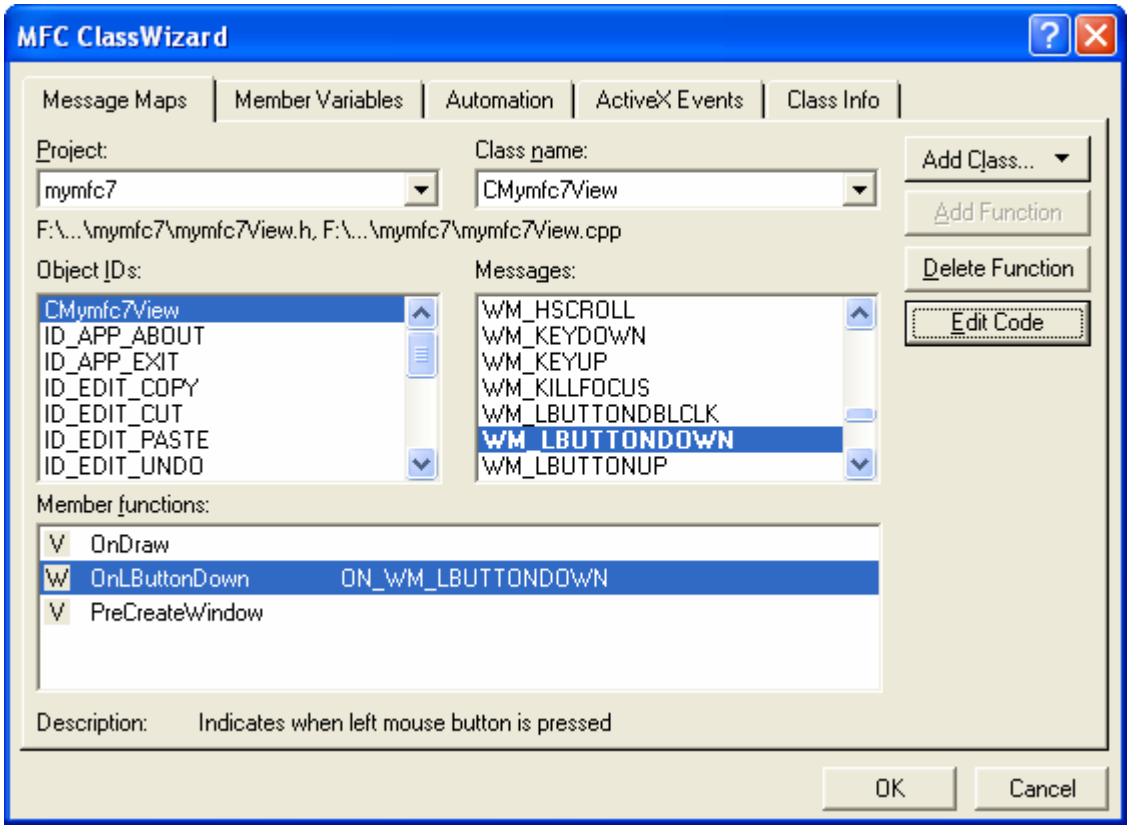


Figure 44: Mapping a `WM_LBUTTONDOWN` (clicking the left mouse button) message to object's ID.

3. Write the code for `OnLButtonDown()` in file **mymfc7View.cpp** by clicking the **Edit Code** button. Add the following code below. Most of the code consists of `TRACE` statements to print the dialog data members after the user exits the dialog. The `CMymfc7Dialog` constructor call and the `DoModal()` call are the critical statements, however:

```
void CMymfc7View::OnLButtonDown(UINT nFlags, CPoint point)
{
    CMymfc7Dialog dlg;
    dlg.m_strName  = "Porter, Harry";
    dlg.m_nSsn     = 12345678;
    dlg.m_nCat     = 1;  // 0 = hourly, 1 = salary
    dlg.m_strBio   = "Like to do programming for fun";
    dlg.m_bInsLife = FALSE;
    dlg.m_bInsDis  = TRUE;
    dlg.m_bInsMed  = TRUE;
    dlg.m_strDept  = "Documentation";
    dlg.m_strSkill = "Executive";
    dlg.m_nLang    = 1;
    dlg.m_strEduc  = "High School";
    dlg.m_nLoyal   = dlg.m_nRely = 50;

   int ret = dlg.DoModal();

    TRACE("DoModal return = %d\n", ret);
    TRACE("name = %s, ssn = %d, cat = %d\n", dlg.m_strName, dlg.m_nSsn, dlg.m_nCat);
    TRACE("dept = %s, skill = %s, lang = %d, educ = %s\n",
          dlg.m_strDept, dlg.m_strSkill, dlg.m_nLang, dlg.m_strEduc);
    TRACE("life = %d, dis = %d, med = %d, bio = %s\n",
          dlg.m_bInsLife, dlg.m_bInsDis, dlg.m_bInsMed, dlg.m_strBio);
    TRACE("loyalty = %d, reliability = %d\n", dlg.m_nLoyal, dlg.m_nRely);
}
```

```
void CMymfc7View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CMymfc7Dialog dlg;
    dlg.m_strName  = "Porter, Harry";
    dlg.m_nSsn     = 12345678;
    dlg.m_nCat     = 1;  // 0 = hourly, 1 = salary
    dlg.m_strBio   = "Like to do programming for fun";
    dlg.m_bInsLife = FALSE;
    dlg.m_bInsDis  = TRUE;
    dlg.m_bInsMed  = TRUE;
    dlg.m_strDept  = "Documentation";
    dlg.m_strSkill = "Executive";
    dlg.m_nLang    = 1;
    dlg.m_strEduc  = "High School";
    dlg.m_nLoyal   = dlg.m_nRely = 50;

   int ret = dlg.DoModal();

    TRACE("DoModal return = %d\n", ret);
    TRACE("name = %s, ssn = %d, cat = %d\n",
          dlg.m_strName, dlg.m_nSsn, dlg.m_nCat);
    TRACE("dept = %s, skill = %s, lang = %d, educ = %s\n",
          dlg.m_strDept, dlg.m_strSkill, dlg.m_nLang, dlg.m_strEduc);
    TRACE("life = %d, dis = %d, med = %d, bio = %s\n",
          dlg.m_bInsLife, dlg.m_bInsDis, dlg.m_bInsMed, dlg.m_strBio);
    TRACE("loyalty = %d, reliability = %d\n", dlg.m_nLoyal, dlg.m_nRely);
}
```

Listing 3.

Add code to the virtual `OnDraw()` function in file **mymfc7View.cpp**. To prompt the user to press the left mouse button, code the `CMymfc7View::OnDraw` function. The skeleton was generated by AppWizard. The following code (which you type in) replaces the existing code:

```
void CMymfc7View::OnDraw(CDC* pDC)
{
    pDC->TextOut(50, 50, "Press the left mouse button to launch the funny dialog
box.");
}
```

```
void CMymfc7View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    pDC->TextOut(50, 50, "Press the left mouse button to launch the funny dialog box.");
}
```

Listing 4.

To **mymfc7View.cpp**, add the dialog class include statement. The `OnLButtonDown()` function above depends on the declaration of class `CMymfc7Dialog`. You must insert the following `#include` statement:

```
#include "mymfc7Dialog.h"
```

at the top of the `CMymfc7View` class source code file (**mymfc7View.cpp**), after the statement:

```
#include "mymfc7View.h"
```

```
// mymfc7View.cpp : implementation

#include "stdafx.h"
#include "mymfc7.h"

#include "mymfc7Doc.h"
#include "mymfc7View.h"
#include "mymfc7Dialog.h"

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

Listing 5.

Build and test the application. If you have done everything correctly, you should be able to build and run the MYMFC7 application through Visual C++. Try entering data in each control, and then click the **OK** button and observe the TRACE results in the **Debug** window. Notice that the scroll bar controls don't do much yet; we'll attend to them later. Notice what happens when you press Enter while typing in text data in a control: the dialog closes immediately.

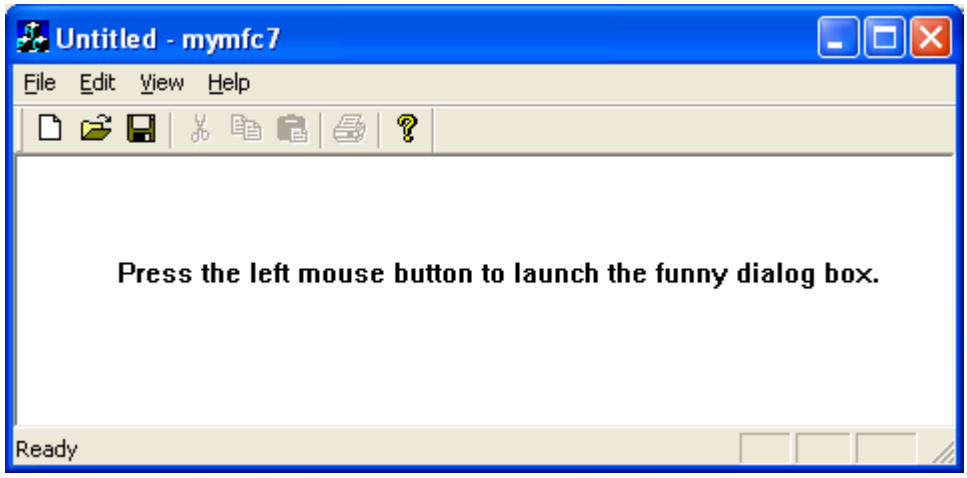


Figure 45: MYMFC7 program output.

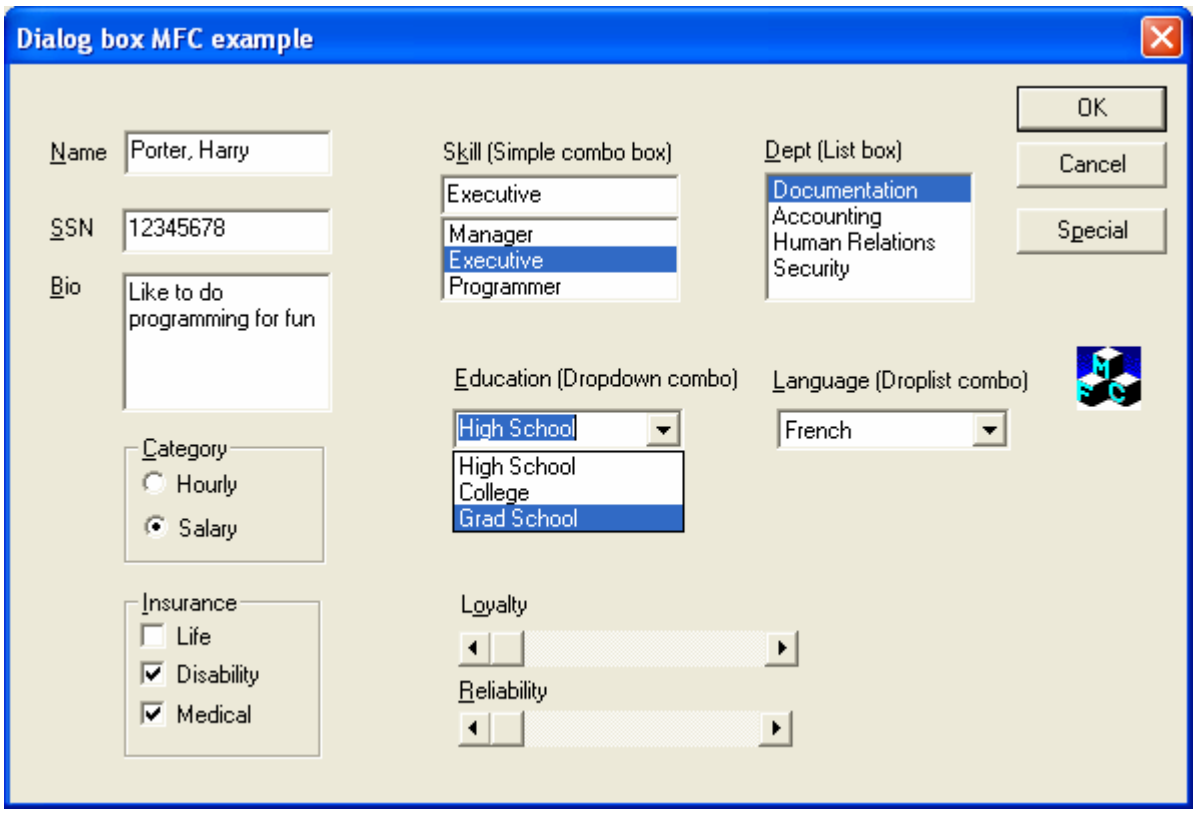


Figure 46: MYMFC7 program output when the left mouse button is clicked, full of controls.

Notice that our horizontal scroll bar is not working at this moment. Don't worry, it is our next task.

Understanding the MYMFC7 Application

When your program calls `DoModal()`, control is returned to your program only when the user closes the dialog. If you understand that, you understand modal dialogs. When you start creating modeless dialogs, you'll begin to appreciate the programming simplicity of modal dialogs. A lot happens "out of sight" as a result of that `DoModal()` call, however. Here's a "what calls what" summary:

```
CDialog::DoModal
    CMymfc7Dialog::OnInitDialog
        …additional initialization…
        CDialog::OnInitDialog
            CWnd::UpdateData(FALSE)
                CMymfc7Dialog::DoDataExchange
    user enters data…
    user clicks the OK button
    CMymfc7Dialog::OnOK
        …additional validation…
        CDialog::OnOK
            CWnd::UpdateData(TRUE)
                CMymfc7Dialog::DoDataExchange
            CDialog::EndDialog(IDOK)
```

`OnInitDialog()` and `DoDataExchange()` are virtual functions overridden in the `CMymfc7Dialog` class. Windows calls `OnInitDialog()` as part of the dialog initialization process, and that results in a call to `DoDataExchange()`, a `CWnd` virtual function that was overridden by ClassWizard. Here is a listing of that function:

```
void CMymfc7Dialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CMymfc7Dialog)
    DDX_Text(pDX, IDC_BIO, m_strBio);
    DDX_Radio(pDX, IDC_CAT, m_nCat);
    DDX_LBString(pDX, IDC_DEPT, m_strDept);
    DDX_Check(pDX, IDC_DIS, m_bInsDis);
    DDX_CBString(pDX, IDC_EDU, m_strEduc);
    DDX_CBIndex(pDX, IDC_LANG, m_nLang);
    DDX_Check(pDX, IDC_LIFE, m_bInsLife);
    DDX_Scroll(pDX, IDC_LOYAL, m_nLoyal);
    DDX_Check(pDX, IDC_MED, m_bInsMed);
    DDX_Text(pDX, IDC_NAME, m_strName);
    DDX_Scroll(pDX, IDC_RELY, m_nRely);
    DDX_CBString(pDX, IDC_SKILL, m_strSkill);
    DDX_Text(pDX, IDC_SSN, m_nSsn);
    //}}AFX_DATA_MAP
}
```

Listing 6.

The `DoDataExchange()` function and the `DDX_` (exchange) and `DDV_` (validation) functions are "bidirectional." If `UpdateData()` is called with a `FALSE` parameter, the functions transfer data from the data members to the dialog controls. If the parameter is `TRUE`, the functions transfer data from the dialog controls to the data members. `DDX_Text` is overloaded to accommodate a variety of data types. The `EndDialog()` function is critical to the dialog exit procedure. `DoModal()` returns the parameter passed to `EndDialog()`. `IDOK` accepts the dialog's data, and `IDCANCEL` cancels the dialog. You can write your own "custom" `DDX` function and wire it into Visual C++. This feature is useful if you're using a unique data type throughout your application.

Enhancing the Dialog Program

The MYMFC7 program required little coding for a lot of functionality. Now we'll make a new version of this program that uses some hand-coding to add extra features. We'll eliminate MYMFC7's rude habit of dumping the user in response to a press of the **Enter** key, and we'll hook up the scroll bar controls.

Taking Control of the `OnOK()` Exit

In the original MYMFC7 program, the `CDialog::OnOK` virtual function handled the **OK** button, which triggered data exchange and the exit from the dialog. Pressing the **Enter** key happens to have the same effect, and that might or might not be what you want. If the user presses **Enter** while in the Name edit control, for example, the dialog closes immediately. What's going on here? When the user presses **Enter**, Windows looks to see which pushbutton has the input focus, as indicated on the screen by a dotted rectangle. If no button has the focus, Windows looks for the default pushbutton that the program or the resource specifies. The default pushbutton has a thicker border. If the dialog has no default button, the virtual `OnOK()` function is called, even if the dialog does not contain an **OK** button. You can disable the Enter key by writing a do-nothing `CMymfc7Dialog::OnOK` function and adding the exit code to a new function that responds to clicking the **OK** button. Here are the steps:

1. Use ClassWizard to "map" the `IDOK` button to the virtual `OnOK()` function. In ClassWizard, choose `IDOK` from the `CMymfc7Dialog` Object IDs list, and then double-click on `BN_CLICKED`. This generates the prototype and skeleton for `OnOK()`.

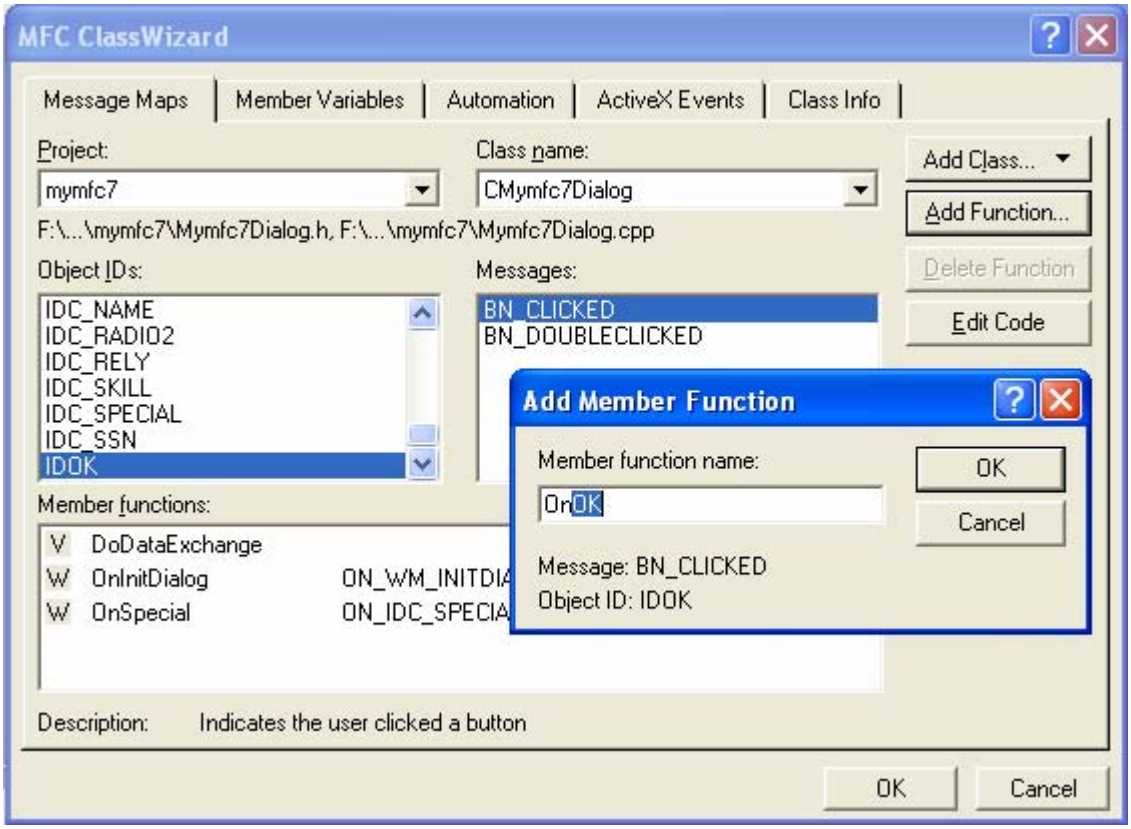


Figure 47: Mapping the `IDOK` button to the virtual `OnOK()` function.

2. Use the dialog editor to change the **OK** button ID. Select the OK button, change its ID from `IDOK` to `IDC_OK`, and then uncheck its **Default Button** property. Leave the `OnOK()` function alone.

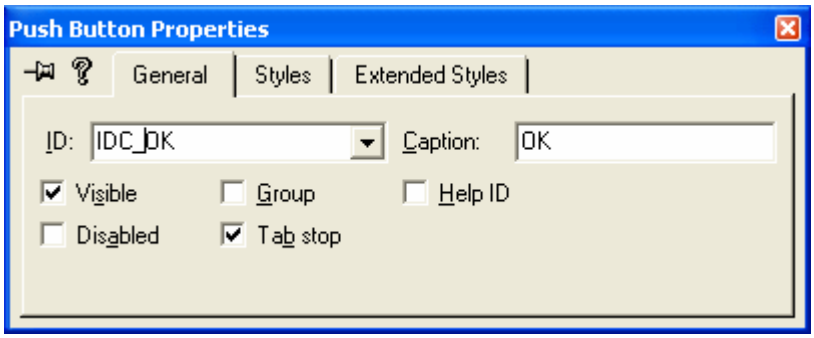


Figure 48: Changing the **OK** push button ID.

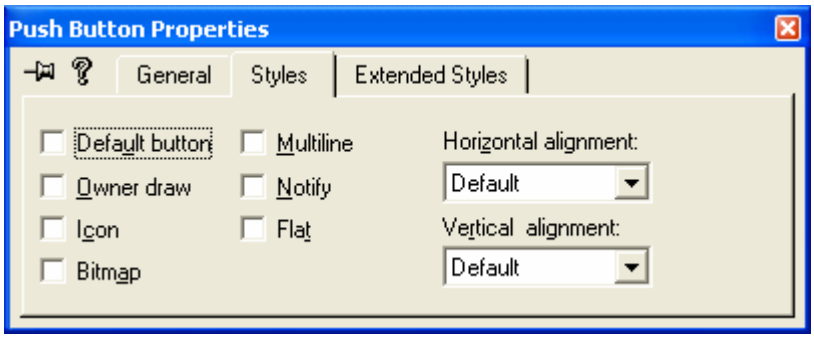


Figure 49: Modifying the **OK** push button properties.

3. Use ClassWizard to create a member function called `OnClickedOk()`. This `CMymfc7Dialog` class member function is keyed to the `BN_CLICKED` message from the newly renamed control `IDC_OK`.

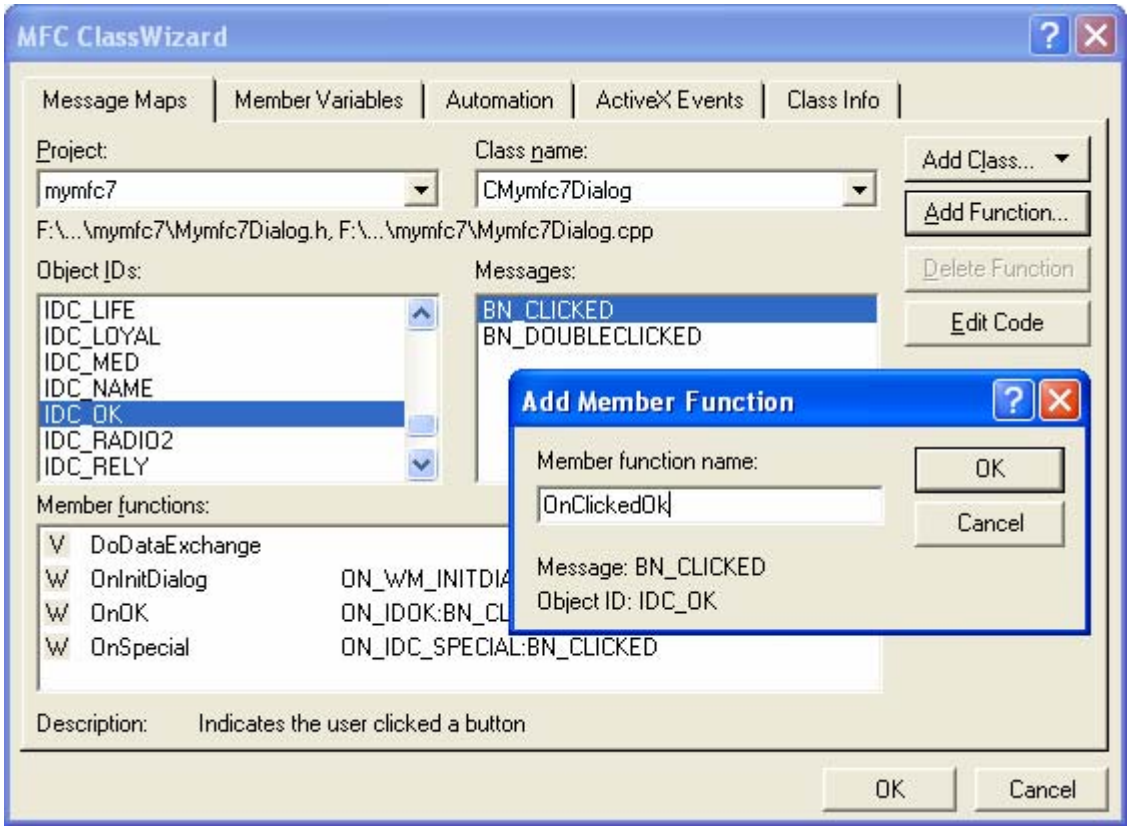


Figure 50: Creating an `OnClickedOk()` member function and maps it to `IDC_OK` ID.

4. Edit the body of the `OnClickedOk()` function in **mymfc7Dialog.cpp**. This function calls the base class `OnOK()` function, as did the original `CMymfc7Dialog::OnOK` function. Here is the code:

```
void CMymfc7Dialog::OnClickedOk()
{
    TRACE("CMymfc7Dialog::OnClickedOk\n");
    CDialog::OnOK();
}
```

```
void CMymfc7Dialog::OnClickedOk()
{
    // TODO: Add your control notification handler code here
    TRACE("CMymfc7Dialog::OnClickedOk\n");
    CDialog::OnOK();
}
```

Listing 7.

Edit the original `OnOK()` function in **mymfc7Dialog.cpp**. This function is a "leftover" handler for the old `IDOK` button. Edit the code as shown here:

```
void CMymfc7Dialog::OnOK()
{
    // dummy OnOK function -- do NOT call CDialog::OnOK()
    TRACE("CMymfc7Dialog::OnOK\n");
```

```
}

void CMymfc7Dialog::OnOK()
{
    // TODO: Add extra validation here
    // dummy OnOK function -- do NOT call CDialog::OnOK()
    TRACE("CMymfc7Dialog::OnOK\n");
}
```

Listing 8.

Build and test the application. Try pressing the **Enter** key now. Nothing should happen, but TRACE output should appear in the **Debug** window. Clicking the **OK** button should exit the dialog as before, however.

```
...
Loaded 'C:\WINDOWS\system32\mslbui.dll', no matching symbolic information found.
Warning: skipping non-radio button in group.
CMymfc7Dialog::OnClickedOk
Warning: skipping non-radio button in group.
DoModal return = 1
name = Porter, Harry, ssn = 12345678, cat = 1
dept = Documentation, skill = Executive, lang = 1, educ = High School
life = 0, dis = 1, med = 1, bio = Like to do programming for fun
loyalty = 0, reliability = 0
The thread 0xF64 has exited with code 0 (0x0).
The program 'F:\mfcproject\mymfc7\Debug\mymfc7.exe' has exited with code 0 (0x0).
```

For Win32 Programmers

Dialog controls send `WM_COMMAND` notification messages to their parent dialogs. For a single button click, for example, the bottom 16 bits of `wParam` contain the button ID, the top 16 bits of `wParam` contain the `BN_CLICKED` notification code, and `lParam` contains the button handle. Most window procedure functions process these notification messages with a nested switch statement. MFC "flattens out" the message processing logic by "promoting" control notification messages to the same level as other Windows messages. For a **Delete** button (for example), ClassWizard generates notification message map entries similar to these:

```
ON_BN_CLICKED(IDC_DELETE, OnDeleteClicked)
ON_BN_DOUBLECLICKED(IDC_DELETE, OnDeleteDblClicked)
```

Button events are special because they generate command messages if your dialog class doesn't have notification handlers like the ones above. As Module 13 explains, the application framework "routes" these command messages to various objects in your application. You could also map the control notifications with a more generic `ON_COMMAND` message-handling entry like this:

```
ON_COMMAND(IDC_DELETE, OnDelete)
```

In this case, the `OnDelete()` function is unable to distinguish between a single click and a double click, but that's no problem because few Windows-based programs utilize double clicks for buttons.

`OnCancel()` Processing

Just as pressing the Enter key triggers a call to `OnOK()`, pressing the **Esc** key triggers a call to `OnCancel()`, which results in an exit from the dialog with a `DoModal()` return code of `IDCANCEL`. MYMFC7 does no special processing for `IDCANCEL`; therefore, pressing the **Esc** key (or clicking the **Close** button) closes the dialog. You can circumvent this process by substituting a dummy `OnCancel()` function, following approximately the same procedure you used for the **OK** button.

Hooking Up the Scroll Bar Controls

The dialog editor allows you to include scroll bar controls in your dialog, and ClassWizard lets you add integer data members. You must add code to make the **Loyalty** and **Reliability** scroll bars work. Scroll bar controls have position and range values that can be read and written. If you set the range to (0, 100), for example, a corresponding data member with a value of 50 positions the scroll box at the center of the bar. The function `CScrollBar::SetScrollPos` also sets the scroll box position. The scroll bars send the `WM_HSCROLL` and `WM_VSCROLL` messages to the dialog when the user drags the scroll box or clicks the arrows. The dialog's message handlers must decode these messages and position the scroll box accordingly.
Each control you've seen so far has had its own individual message handler function. Scroll bar controls are different because all horizontal scroll bars in a dialog are tied to a single `WM_HSCROLL` message handler and all vertical scroll bars are tied to a single `WM_VSCROLL` handler. Because this monster dialog contains two horizontal scroll bars, the single `WM_HSCROLL` message handler must figure out which scroll bar sent the scroll message. Here are the steps for adding the scroll bar logic to MYMFC7:

Add the class `enum` statements for the minimum and maximum scroll range. In **mymfc7Dialog.h**, add the following lines at the top of the class declaration:

```
enum { nMin = 0 };
enum { nMax = 100 };
```

```
// CMymfc7Dialog dialog

class CMymfc7Dialog : public CDialog
{

    enum { nMin = 0 };
    enum { nMax = 100 };

// Construction
```

Listing 9.

Edit the `OnInitDialog()` function to initialize the scroll ranges. In the `OnInitDialog()` function, we'll set the minimum and the maximum scroll values such that the `CMymfc7Dialog` data members represent percentage values. A value of 100 means "Set the scroll box to the extreme right"; a value of 0 means "Set the scroll box to the extreme left." Add the following code to the `CMymfc7Dialog` member function `OnInitDialog()` in the file **mymfc7Dialog.cpp**:

```
CScrollBar* pSB = (CScrollBar*) GetDlgItem(IDC_LOYAL);
pSB->SetScrollRange(nMin, nMax);

pSB = (CScrollBar*) GetDlgItem(IDC_RELY);
pSB->SetScrollRange(nMin, nMax);
```

```
BOOL CMymfc7Dialog::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: Add extra initialization here
    // Be careful to call CDialog::OnInitDialog
    // only once in this function
    CListBox* pLB = (CListBox*) GetDlgItem(IDC_DEPT);
    pLB->InsertString(-1, "Documentation");
    pLB->InsertString(-1, "Accounting");
    pLB->InsertString(-1, "Human Relations");
    pLB->InsertString(-1, "Security");

    CScrollBar* pSB = (CScrollBar*) GetDlgItem(IDC_LOYAL);
    pSB->SetScrollRange(nMin, nMax);
    pSB = (CScrollBar*) GetDlgItem(IDC_RELY);
    pSB->SetScrol void __thiscall CScrollBar::SetScrollRange(int,int,int)

    // Call after initialization
    return CDialog::OnInitDialog();
}
```

Listing 10.

Use ClassWizard to add a scroll bar message handler to `CMymfc7Dialog`. Choose the `WM_HSCROLL` message, and then add the member function `OnHScroll()`.

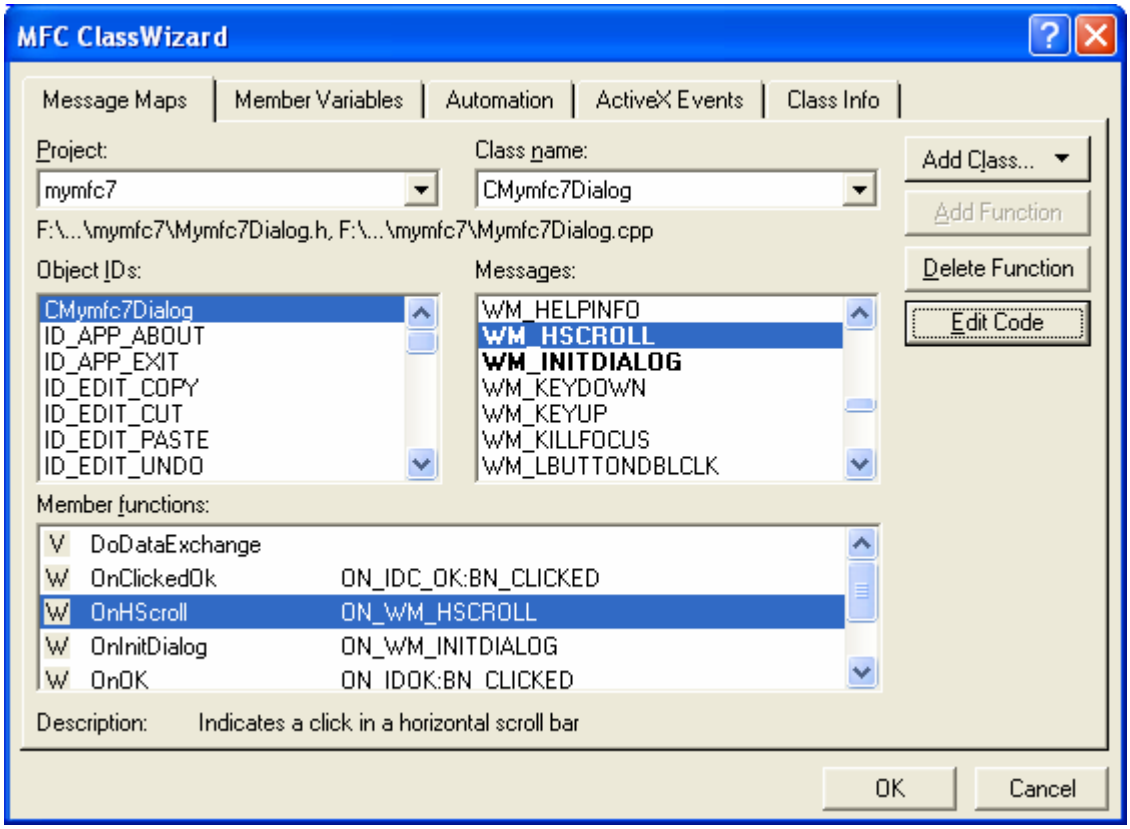


Figure 51: Adding a scroll bar message handler to `CMymfc7Dialog`.

Click the **Edit Code** button and enter the following code:

```
void CMymfc7Dialog::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    int nTemp1, nTemp2;
```

```
    nTemp1 = pScrollBar->GetScrollPos();
    switch(nSBCode) {
    case SB_THUMBPOSITION:
        pScrollBar->SetScrollPos(nPos);
        break;
    case SB_LINELEFT: // left arrow button
        nTemp2 = (nMax - nMin) / 10;
        if ((nTemp1 - nTemp2) > nMin) {
            nTemp1 -= nTemp2;
        }
        else {
            nTemp1 = nMin;
        }
        pScrollBar->SetScrollPos(nTemp1);
        break;
    case SB_LINERIGHT: // right arrow button
        nTemp2 = (nMax - nMin) / 10;
        if ((nTemp1 + nTemp2) < nMax) {
            nTemp1 += nTemp2;
        }
        else {
            nTemp1 = nMax;
        }
        pScrollBar->SetScrollPos(nTemp1);
        break;
    }
}
```

```
void CMymfc7Dialog::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default
    int nTemp1, nTemp2;

    nTemp1 = pScrollBar->GetScrollPos();
    switch(nSBCode) {
    case SB_THUMBPOSITION:
        pScrollBar->SetScrollPos(nPos);
        break;
    case SB_LINELEFT: // left arrow button
        nTemp2 = (nMax - nMin) / 10;
        if ((nTemp1 - nTemp2) > nMin) {
            nTemp1 -= nTemp2;
        }
        else {
            nTemp1 = nMin;
        }
        pScrollBar->SetScrollPos(nTemp1);
        break;
    case SB_LINERIGHT: // right arrow button
        nTemp2 = (nMax - nMin) / 10;
        if ((nTemp1 + nTemp2) < nMax) {
            nTemp1 += nTemp2;
        }
        else {
            nTemp1 = nMax;
        }
        pScrollBar->SetScrollPos(nTemp1);
        break;
    }
}
```

Listing 11.

The scroll bar functions use 16-bit integers for both range and position.

Build and test the application. Build and run MYMFC7 again. Do the scroll bars work this time? The scroll boxes should "stick" after you drag them with the mouse, and they should move when you click the scroll bars' arrows. Notice that we haven't added logic to cover the user's click on the scroll bar itself.

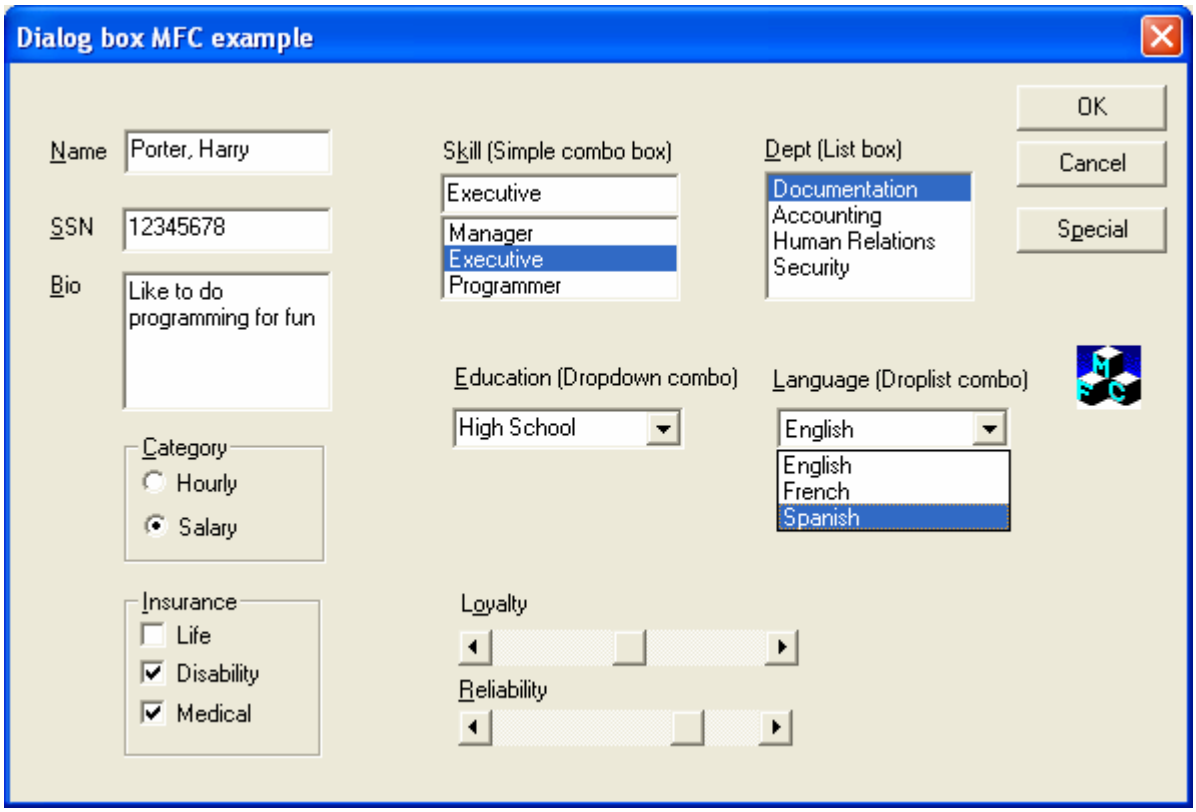


Figure 52: New MYMFC7 program output activating the scroll bars.

Identifying Controls: CWnd Pointers and Control IDs

When you lay out a dialog resource in the dialog editor, you identify each control by an ID such as `IDC_SSN`. In your program code, however, you often need access to a control's underlying window object. The MFC library provides the `CWnd::GetDlgItem` function for converting an ID to a `CWnd` pointer. You've seen this already in the `OnInitDialog()` member function of class `CMymfc7Dialog`. The application framework "manufactured" this returned `CWnd` pointer because there never was a constructor call for the control objects. This pointer is temporary and should not be stored for later use. If you need to convert a `CWnd` pointer to a control ID, use the MFC library `GetDlgCtrlID()` member function of class `CWnd`.

Setting the Color for the Dialog Background and for Controls

You can change the background color of individual dialogs or specific controls in a dialog, but you have to do some extra work. The parent dialog is sent a `WM_CTLCOLOR` message for each control immediately before the control is displayed. A `WM_CTLCOLOR` message is also sent on behalf of the dialog itself. If you map this message in your derived dialog class, you can set the foreground and background text colors and select a brush for the control or dialog non-text area.
The following is a sample `OnCtlColor()` function that sets all edit control **text color** to red and the **dialog background** to yellow. The `m_hYellowBrush` and `m_hRedBrush` variables are data members of type `HBRUSH`, should be initialized in the dialog's `OnInitDialog()` function. The `nCtlColor` parameter indicates the type of control, and the `pWnd` parameter identifies the specific control. If you wanted to set the color for an individual edit control, you would convert `pWnd` to a child window ID and test it.

```
HBRUSH CMyDialog::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
```

```
{
    if (nCtlColor == CTLCOLOR_EDIT) {
        pDC->SetBkColor(RGB(255, 255, 0));  // yellow
        return m_hYellowBrush;
    }
    if (nCtlColor == CTLCOLOR_DLG) {
        pDC->SetBkColor(RGB(255, 0, 0));    // red
        return m_hRedBrush;
    }
    return CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
}
```

The dialog does not post the `WM_CTLCOLOR` message in the message queue; instead, it calls the Win32 `SendMessage()` function to send the message immediately. Thus the message handler can return a parameter, in this case a handle to a brush. This is not an MFC `CBrush` object but rather a Win32 `HBRUSH`. You can create the brush by calling the Win32 functions `CreateSolidBrush()`, `CreateHatchBrush()`, and so forth.

For Win32 Programmers

Actually, Win32 no longer has a `WM_CTLCOLOR` message. It was replaced by control-specific messages such as `WM_CTLCOLORBTN`, `WM_CTLCOLORDLG`, and so on. MFC and ClassWizard process these messages invisibly, so your programs look as though they're mapping the old 16-bit `WM_CTLCOLOR` messages. This trick makes debugging more complex, but it makes portable code easier to write. Another option would be to use the `ON_MESSAGE` macro to map the real Win32 messages. If your dialog class (or other MFC window class) doesn't map the `WM_CTLCOLOR` message, the framework reflects the message back to the control.

Let try this one. Add two public data member, `m_hYellowBrush` and `m_hRedBrush` of type `HBRUSH`.

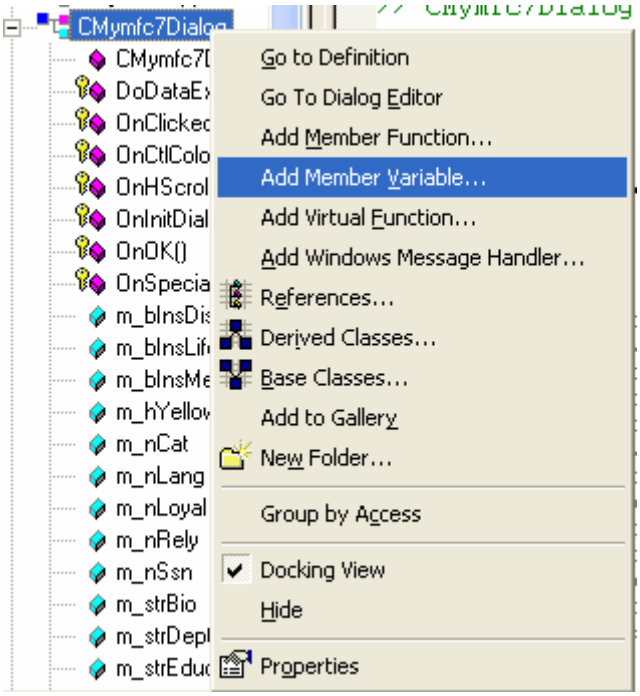


Figure 53: Adding two public data member, `m_hYellowBrush` and `m_hRedBrush` of type `HBRUSH`.

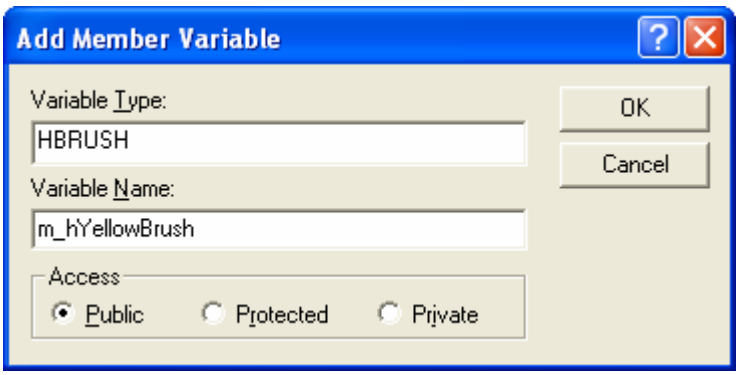


Figure 54: Adding dialog's member variable.

```
// Construction
public:
    HBRUSH m_hRedBrush;
    HBRUSH m_hYellowBrush;
    CMymfc7Dialog(CWnd* pParent = NULL);
```

Listing 12.

Then do the message map for the `WM_CTRLCOLOR` and click the **Edit Code** button.

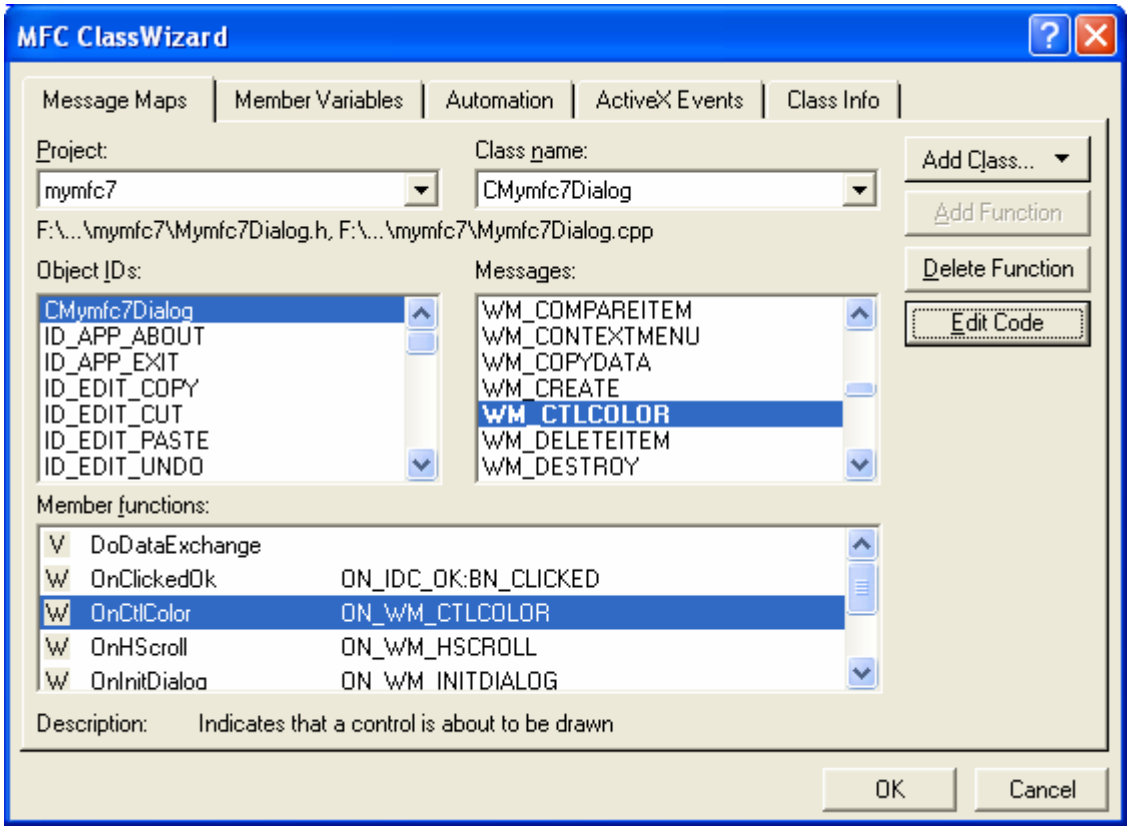


Figure 55: The message map of the `WM_CTRLCOLOR` for `CMymfc7Dialog`.

Enter the following code.

```
HBRUSH CMymfc7Dialog::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
```

```
        // You can try other controls...
        // CTLCOLOR_BTN - Button control
        // CTLCOLOR_DLG - Dialog box
        // CTLCOLOR_EDIT - Edit control
        // CTLCOLOR_LISTBOX - List-box control
        // CTLCOLOR_MSGBOX - Message box
        // CTLCOLOR_SCROLLBAR - Scroll-bar control
        // CTLCOLOR_STATIC - Static control

        if (nCtlColor == CTLCOLOR_EDIT) {
         pDC->SetBkColor(RGB(255, 255, 0)); // yellow edit control background
          pDC->SetTextColor(RGB(255, 0, 0)); // red edit control text

          return m_hYellowBrush;
        }

        if (nCtlColor == CTLCOLOR_BTN) {
         pDC->SetBkColor(RGB(255, 0, 0));    // red for dialog
          return m_hRedBrush;
        }

        return CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

    }
```

Build the program.

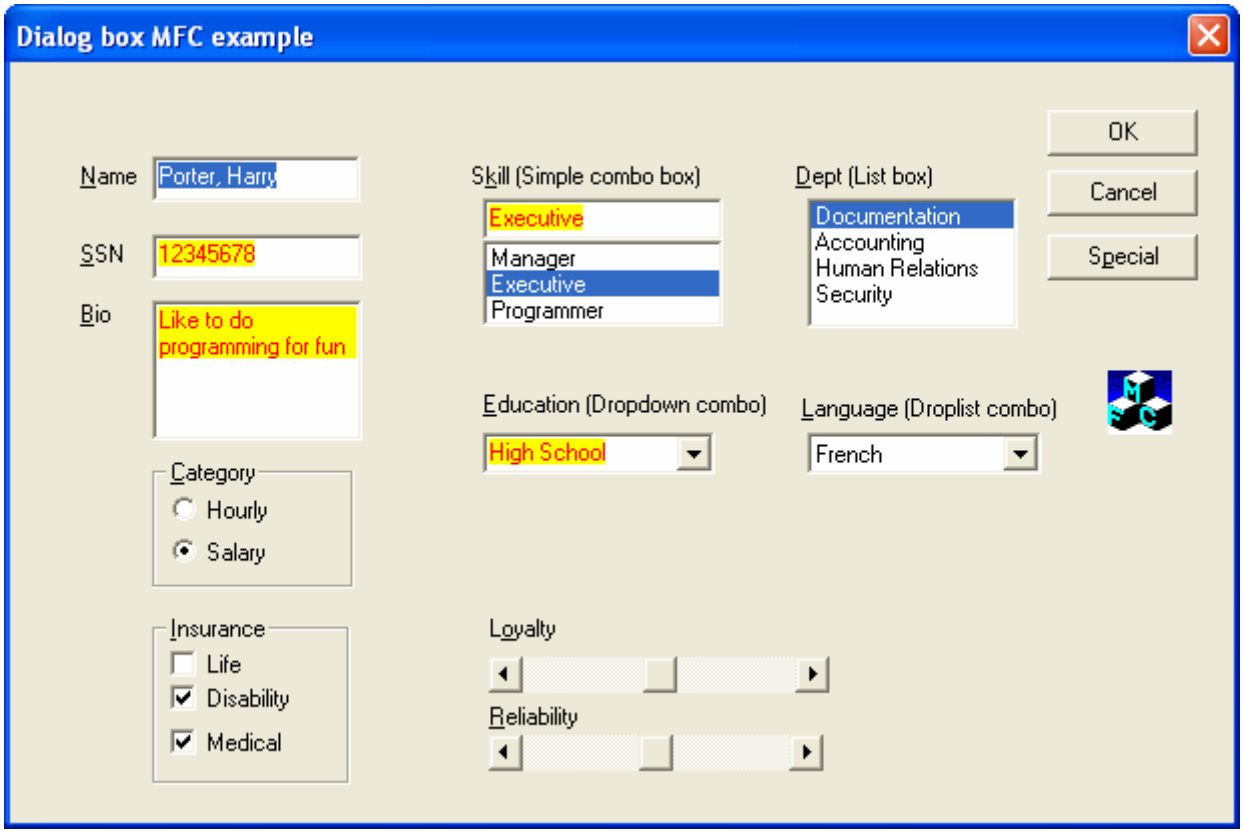


Figure 56: New MYMFC7 program output with all edit control **text color** set to red and the **background** set to yellow.

Painting Inside the Dialog Window

You can paint directly in the client area of the dialog window, but you'll avoid overwriting dialog elements if you paint only inside a control window. If you want to display text only, use the dialog editor to create a blank static control with a

unique ID and then call the `CWnd::SetDlgItemText` function in a dialog member function such as `OnInitDialog()` to place text in the control.
Displaying graphics is more complicated. You must use ClassWizard to add an `OnPaint()` member function to the dialog; this function must convert the static control's ID to a `CWnd` pointer and get its device context. The trick is to draw inside the control window while preventing Windows from overwriting your work later. The `Invalidate()`/`UpdateWindow()` sequence achieves this. Here is an `OnPaint()` function that paints a small black square in a static control:

```
void CMyDialog::OnPaint()
{
    CWnd* pWnd = GetDlgItem(IDC_STATIC1); // IDC_STATIC1 specified
                                          //  in the dialog editor
    CDC* pControlDC = pWnd->GetDC();

    pWnd->Invalidate();
    pWnd->UpdateWindow();
    pControlDC->SelectStockObject(BLACK_BRUSH);
    pControlDC->Rectangle(0, 0, 10, 10);  // black square bullet
    pWnd->ReleaseDC(pControlDC);
}
```

As with all windows, the dialog's `OnPaint()` function is called only if some part of the dialog is invalidated. You can force the `OnPaint()` call from another dialog member function with the following statement:

```
Invalidate();
```

Adding Dialog Controls at Runtime

You've seen how to use the resource editor to create dialog controls at build time. If you need to add a dialog control at runtime, here are the programming steps:

1. Add an embedded control window data member to your dialog class. The MFC control window classes include `CButton`, `CEdit`, `CListBox`, and `CComboBox`. An embedded control C++ object is constructed and destroyed along with the dialog object.
2. Choose **Resource Symbols** from Visual C++'s **View** menu. Add an **ID constant** for the new control.
3. Use ClassWizard to map the `WM_INITDIALOG` message, thus overriding `CDialog::OnInitDialog`. This function should call the embedded control window's `Create()` member function. This call displays the new control in the dialog. Windows will destroy the control window when it destroys the dialog window.
4. In your derived dialog class, manually add the necessary notification message handlers for your new control.

We will learn this more detail in another Module.

Using Other Control Features

You've seen how to customize the control class `CScrollBar` by adding code in the dialog's `OnInitDialog()` member function. You can program other controls in a similar fashion. In the Microsoft Visual C++ MFC Library Reference, or in the online help under "Microsoft Foundation Class Libary and Templates," look at the control classes, particularly `CListBox` and `CComboBox`. Each has a number of features that ClassWizard does not directly support. Some combo boxes, for example, can support multiple selections. If you want to use these features, don't try to use ClassWizard to add data members. Instead, define your own data members and add your own exchange code in `OnInitDialog()` and `OnClickedOK()`.

For Win32 Programmers

If you've programmed controls in Win32, you'll know that parent windows communicate to controls via Windows messages. So what does a function such as `CListBox::InsertString` do? (You've seen this function called in your `OnInitDialog()` function.) If you look at the MFC source code, you'll see that `InsertString()` sends an `LB_INSERTSTRING` message to the designated list-box control. Other control class member functions don't send

messages because they apply to all window types. The `CScrollView::SetScrollRange` function, for example, calls the Win32 `SetScrollRange()` function, specifying the correct `hWnd` as a parameter.

Windows Common Controls

The controls you used in MYMFC7 are great learning controls because they're easy to program. Now you're ready for some more "interesting" controls. We'll take a look at some important new Windows controls, introduced for Microsoft Windows 95 and available in Microsoft Windows NT. These include the progress indicator, trackbar, spin button control, list control, and tree control.
The code for these controls is in the Windows **COMCTL32.DLL** file. This code includes the window procedure for each control, together with code that registers a window class for each control. The registration code is called when the DLL is loaded. When your program initializes a dialog, it uses the symbolic class name in the dialog resource to connect to the window procedure in the DLL. Thus your program owns the control's window, but the code is in the DLL. Except for ActiveX controls, most controls work this way. Example MYMFC8 uses the aforementioned controls. Figure 57 shows the dialog from that example. Refer to it when you read the control descriptions that follow.
Be aware that ClassWizard offers no member variable support for the common controls. You'll have to add code to your `OnInitDialog()` and `OnOK()` functions to initialize and read control data. ClassWizard will, however, allow you to map notification messages from common controls.

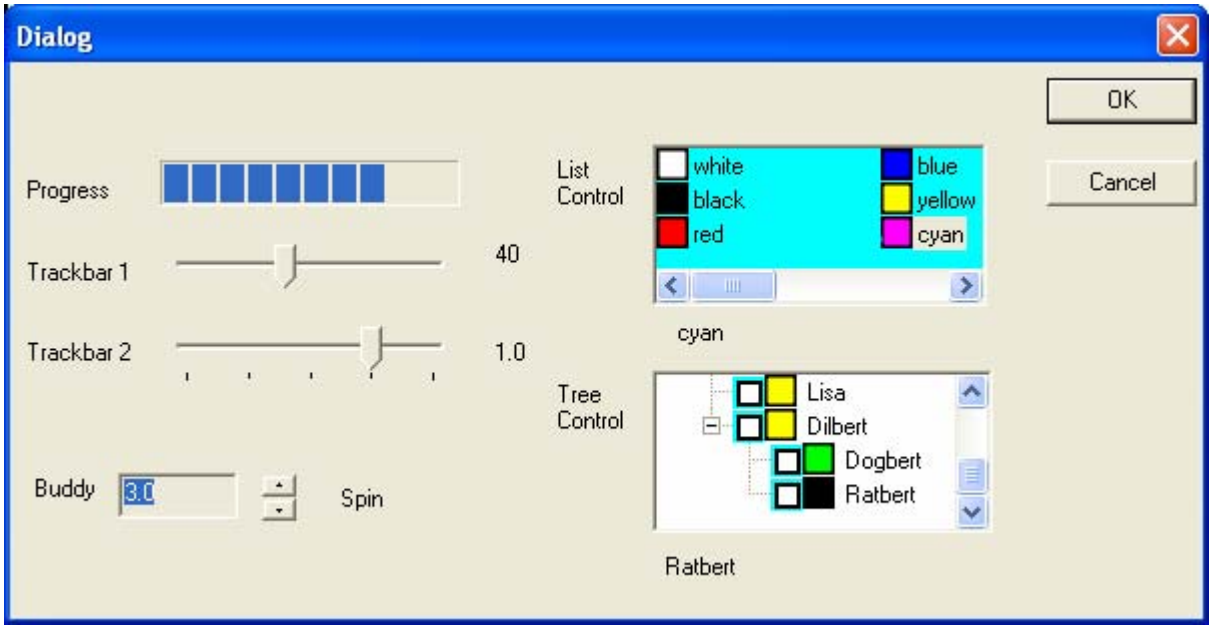


Figure 57: The Windows Common Controls Dialog example.

The Progress Indicator Control

The progress indicator is the easiest common control to program and is represented by the MFC `CProgressCtrl` class. It is generally used only for output. This control, together with the trackbar, can effectively replace the scroll bar controls you saw in the previous example. To initialize the progress indicator, call the `SetRange()` and `SetPos()` member functions in your `OnInitDialog()` function, and then call `SetPos()` anytime in your message handlers. The progress indicator shown in Figure 6-2 has a range of 0 to 100, which is the default range.

The Trackbar Control

The trackbar control (class `CSliderCtrl`), sometimes called a slider, allows the user to set an "analog" value. Trackbars would have been more effective than sliders for **Loyalty** and **Reliability** in the MYMFC7 example. If you specify a large range for this control, 0 to 100 or more, for example, the trackbar's motion appears continuous. If you specify a small range, such as 0 to 5, the tracker moves in discrete increments. You can program tick marks to match the increments. In this discrete mode, you can use a trackbar to set such items as the display screen resolution, lens f-stop values, and so forth. The trackbar does not have a default range.

The trackbar is easier to program than the scroll bar because you don't have to map the WM_HSCROLL or WM_VSCROLL messages in the dialog class. As long as you set the range, the tracker moves when the user slides it or clicks in the body of the trackbar. You might choose to map the scroll messages anyway if you want to show the position value in another control. The GetPos() member function returns the current position value. The top trackbar in Figure 6-2 operates continuously in the range 0 to 100. The bottom trackbar has a range of 0 to 4, and those indexes are mapped to a series of double-precision values (4.0, 5.6, 8.0, 11.0, and 16.0).

The Spin Button Control

The spin button control (class CSpinButtonCtrl) is an itsy-bitsy scroll bar that's most often used in conjunction with an edit control. The edit control, located just ahead of the spin control in the dialog's tabbing order, is known as the spin control's buddy. The idea is that the user holds down the left mouse button on the spin control to raise or lower the value in the edit control. The spin speed accelerates as the user continues to hold down the mouse button.
If your program uses an integer in the buddy, you can avoid C++ programming almost entirely. Just use ClassWizard to attach an integer data member to the edit control, and set the spin control's range in the OnInitDialog() function. You probably won't want the spin control's default range, which runs backward from a minimum of 100 to a maximum of 0. Don't forget to select **Auto Buddy** and **Set Buddy Integer** in the spin control's Styles property page. You can call the SetRange() and SetAccel() member functions in your OnInitDialog() function to change the range and the acceleration profile. If you want your edit control to display a non-integer, such as a time or a floating-point number, you must map the spin control's WM_VSCROLL (or WM_HSCROLL) messages and write handler code to convert the spin control's integer to the buddy's value.

The List Control

Use the list control (class CListCtrl) if you want a list that contains images as well as text. Figure 6-2 shows a list control with a "list" view style and small icons. The elements are arranged in a grid, and the control includes horizontal scrolling. When the user selects an item, the control sends a notification message, which you map in your dialog class. That message handler can determine which item the user selected. Items are identified by a zero-based integer index.
Both the list control and the tree control get their graphic images from a common control element called an image list (class CImageList). Your program must assemble the image list from icons or bitmaps and then pass an image list pointer to the list control. Your OnInitDialog() function is a good place to create and attach the image list and to assign text strings. The InsertItem() member function serves this purpose. List control programming is straightforward if you stick with strings and icons. If you implement drag and drop or if you need custom owner-drawn graphics, you've got more work to do.

The Tree Control

You're already familiar with tree controls if you've used Microsoft Windows Explorer or Visual C++'s Workspace view. The MFC CTreeCtrl class makes it easy to add this same functionality to your own programs. Figure 6-2 illustrates a tree control that shows a modern American combined family. The user can expand and collapse elements by clicking the + and - buttons or by double-clicking the elements. The icon next to each item is programmed to change when the user selects the item with a single click.
The list control and the tree control have some things in common: they can both use the same image list, and they share some of the same notification messages. Their methods of identifying items are different, however. The tree control uses an HTREEITEM handle instead of an integer index. To insert an item, you call the InsertItem() member function, but first you must build up a TV_INSERTSTRUCT structure that identifies (among other things) the string, the image list index, and the handle of the parent item (which is null for top-level items). As with list controls, infinite customization possibilities are available for the tree control. For example, you can allow the user to edit items and to insert and delete items.

The WM_NOTIFY Message

The original Windows controls sent their notifications in WM_COMMAND messages. The standard 32-bit wParam and lParam message parameters are not sufficient, however, for the information that a common control needs to send to its parent. Microsoft solved this "bandwidth" problem by defining a new message, WM_NOTIFY. With the WM_NOTIFY message, wParam is the control ID and lParam is a pointer to an NMHDR structure, which is managed by the control. This C structure is defined by the following code:

```
typedef struct tagNMHDR {
    HWND hwndFrom; // handle to control sending the message
    UINT idFrom;   // ID of control sending the message
    UINT code;     // control-specific notification code
} NMHDR;
```

Many controls, however, send `WM_NOTIFY` messages with pointers to structures larger than `NMHDR`. Those structures contain the three members above plus appended control-specific members. Many tree control notifications, for example, pass a pointer to an `NM_TREEVIEW` structure that contains `TV_ITEM` structures, a drag point, and so forth. When ClassWizard maps a `WM_NOTIFY` message, it generates a pointer to the appropriate structure.

The MYMFC8 Example

The steps using common controls are shown below.

Run AppWizard to generate the MYMFC8 project. Choose **New** from Visual C++'s **File** menu, and then select **Microsoft AppWizard (exe)** from the **Projects** page. Accept all the defaults but two: select **Single Document** and deselect **Printing And Print Preview**. The options and the default class names are shown here.

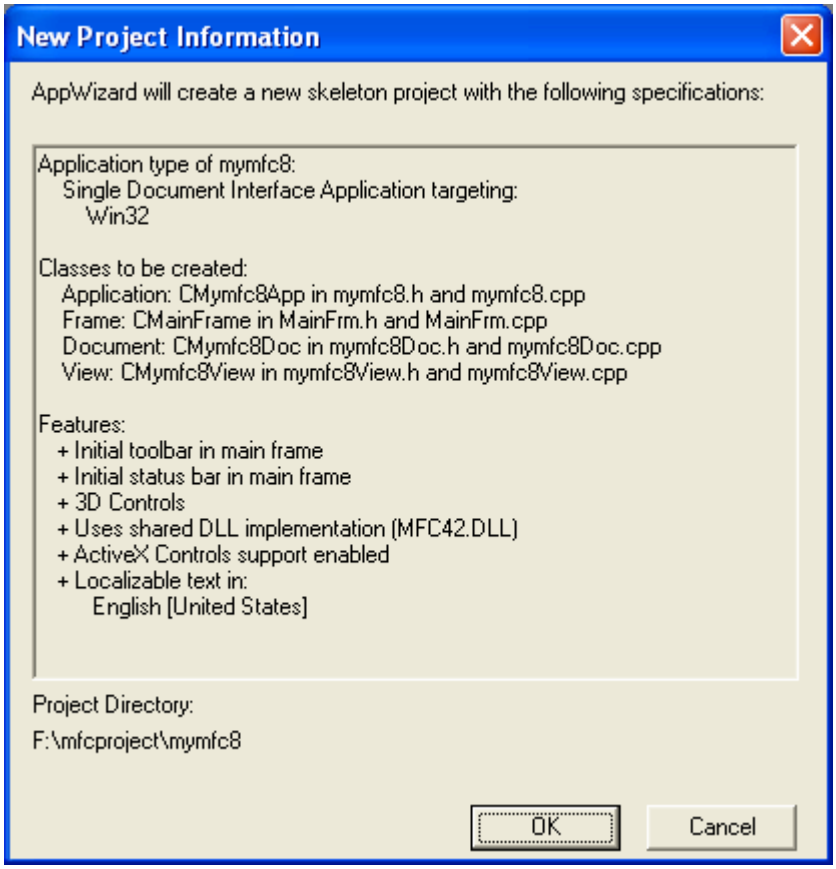


Figure 58: MYMFC8 project summary.

Create a new dialog resource with ID `IDD_DIALOG1` (default ID used). Place the controls as shown in Figure 57. You can select, drag and drop the controls from the control palette. The following table lists the control types and their IDs. Don't worry about the other properties now; you'll set those in the following steps. Some controls might look different than they do in Figure 59 until you set their properties. Set the **tab order** as shown next.

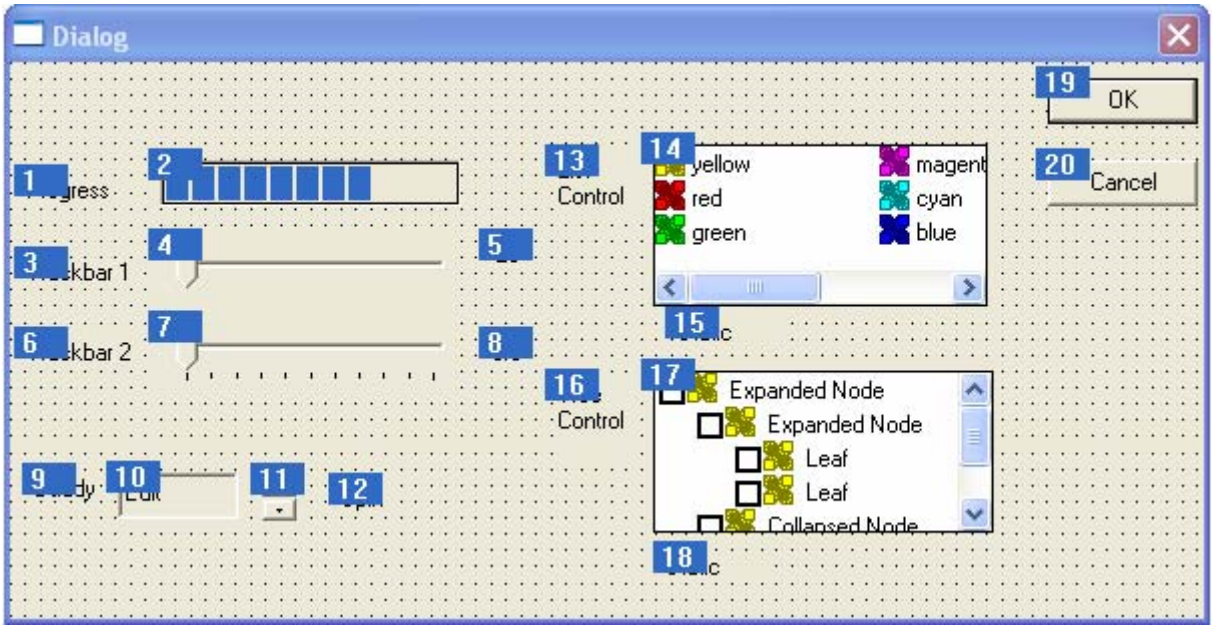


Figure 59: MYMFC8 dialog resource with its common controls.

Tab Sequence	Control Type	Child Window ID
1	Static	IDC_STATIC
2	Progress	IDC_PROGRESS1
3	Static	IDC_STATIC
4	Trackbar (Slider)	IDC_TRACKBAR1
5	Static	IDC_STATIC_TRACK1
6	Static	IDC_STATIC
7	Trackbar (Slider)	IDC_TRACKBAR2
8	Static	IDC_STATIC_TRACK2
9	Static	IDC_STATIC
10	Edit	IDC_BUDDY_SPIN1
11	Spin	IDC_SPIN1
12	Static	IDC_STATIC
13	Static	IDC_STATIC
14	List control	IDC_LISTVIEW1
15	Static	IDC_STATIC_LISTVIEW1
16	Static	IDC_STATIC
17	Tree control	IDC_TREEVIEW1
18	Static	IDC_STATIC_TREEVIEW1
19	Pushbutton	IDOK
20	Pushbutton	IDCANCEL

Table 2: MYMFC8 common controls, IDs and their tab order.

Use ClassWizard to create a new class, `CMymfc8Dialog`, derived from `CDialog`. ClassWizard will automatically prompt you to create this class because it knows that the `IDD_DIALOG1` resource exists without an associated C++ class.

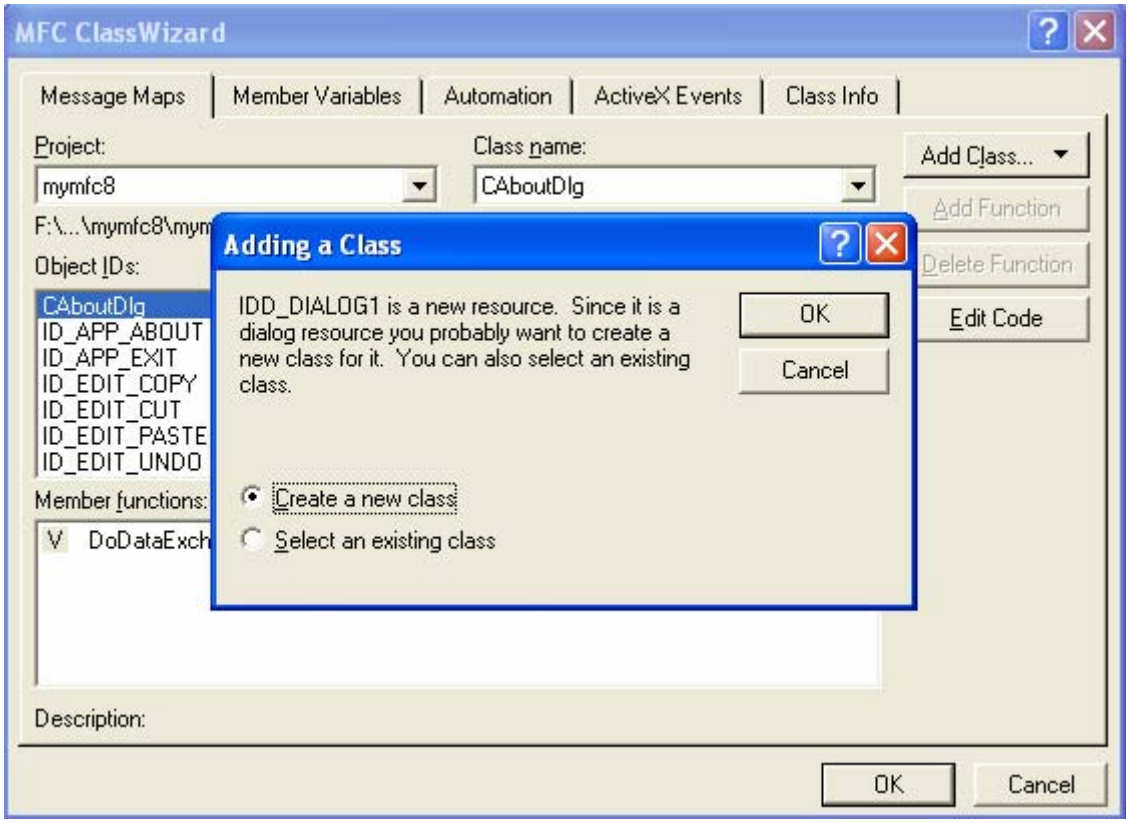


Figure 60: Creating a new class, `CMymfc8Dialog`, derived from `CDialog`.

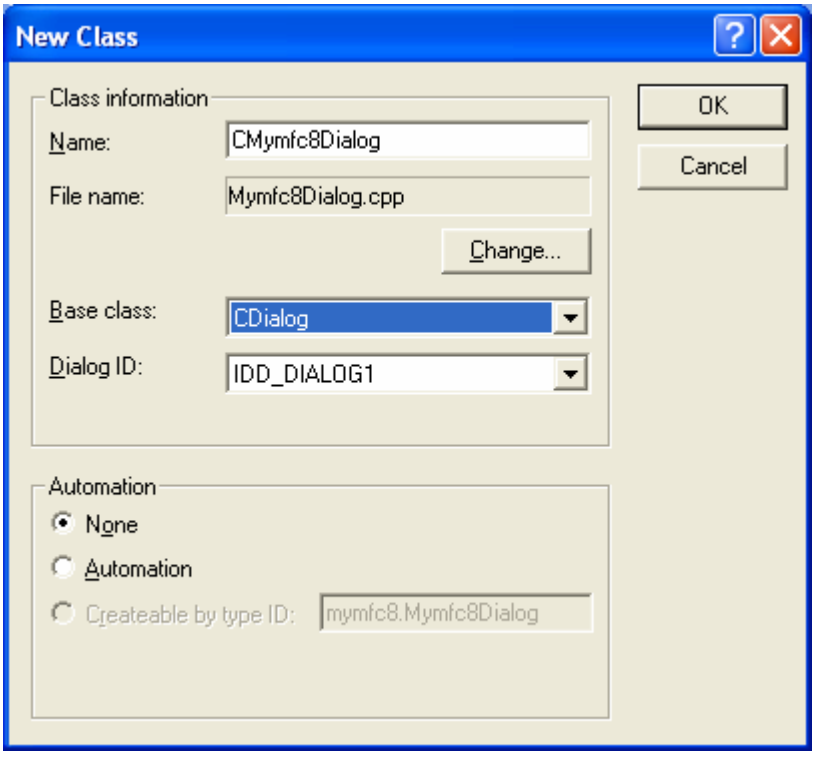


Figure 61: The `CMymfc8Dialog` class information.

Map the `WM_INITDIALOG` message, the `WM_HSCROLL` message, and the `WM_VSCROLL` message.

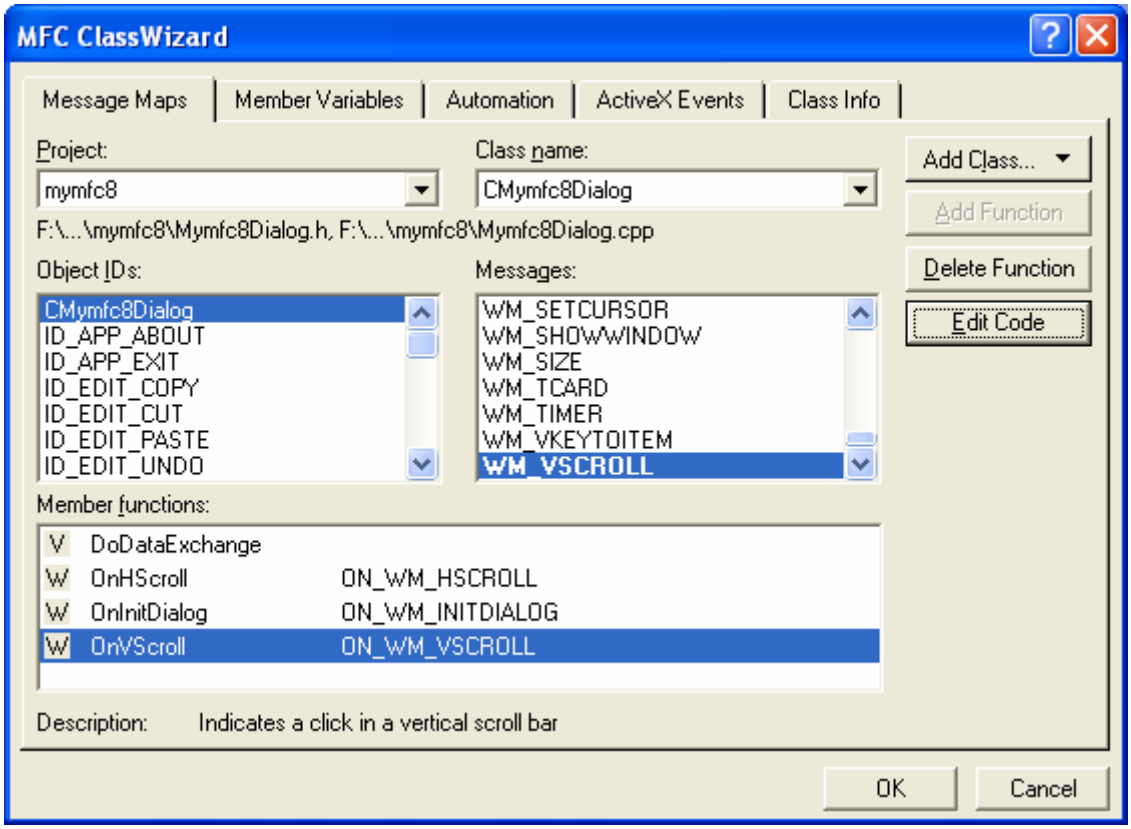


Figure 62: Mapping the `WM_INITDIALOG`, `WM_HSCROLL` and `WM_VSCROLL` messages.

Program the progress control. Because ClassWizard won't generate a data member for this control, you must do it yourself. Add a public integer data member named `m_nProgress` in the `CMymfc8Dialog` class header, and set it to 0 in the constructor.

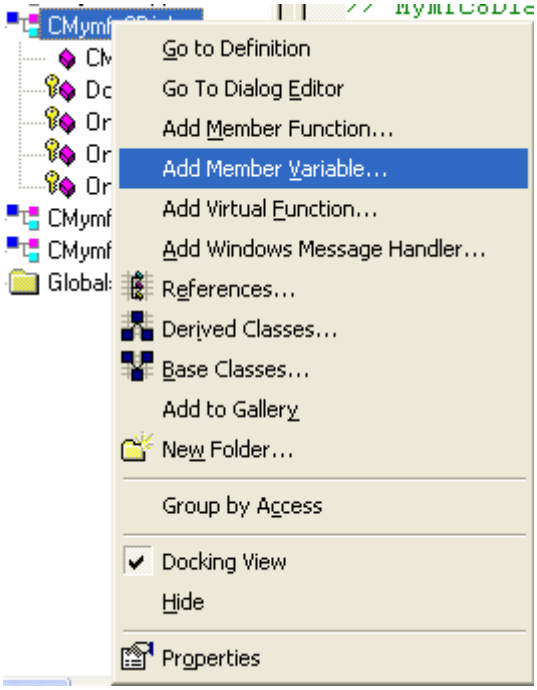


Figure 63: Adding a public integer data member named `m_nProgress`.

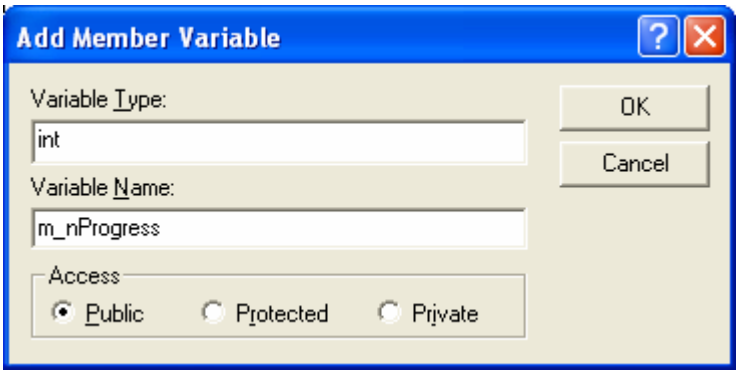


Figure 64: Adding the variable type and name.

```
class CMymfc8Dialog : public CDialog
{
// Construction
public:
    int m_nProgress;
    CMymfc8Dialog(CWnd* pParent = NULL);

// Dialog Data
```

```
// CMymfc8Dialog dialog

CMymfc8Dialog::CMymfc8Dialog(CWnd* pParent /*=NULL*/)
    : CDialog(CMymfc8Dialog::IDD, pParent)
{
    m_nProgress = 0;

    //{{AFX_DATA_INIT(CMymfc8Dialog)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}
```

Listing 13.

Also, add the following code in the `OnInitDialog()` member function:

```
      CProgressCtrl* pProg = (CProgressCtrl*) GetDlgItem(IDC_PROGRESS1);
      pProg->SetRange(0, 100);
      pProg->SetPos(m_nProgress);
```

```
BOOL CMymfc8Dialog::OnInitDialog()
{
    // TODO: Add extra initialization here
    CProgressCtrl* pProg = (CProgressCtrl*) GetDlgItem(IDC_PROGRESS1);
    pProg->SetRange(0, 100);
    pProg->SetPos(m_nProgress);
}
```

Listing 14.

Program the "continuous" trackbar control. Add a public integer data member named `m_nTrackbar1` to the `CMymfc8Dialog` header, and set it to **0** in the constructor.

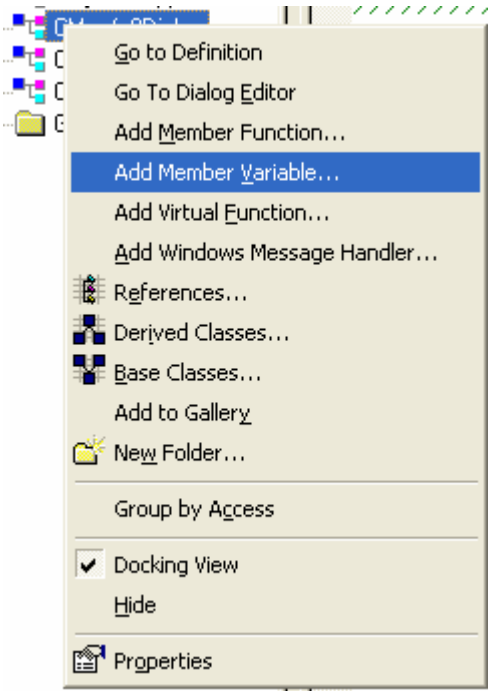


Figure 65: Adding a public integer data member named `m_nTrackbar1` to the `CMymfc8Dialog` header.

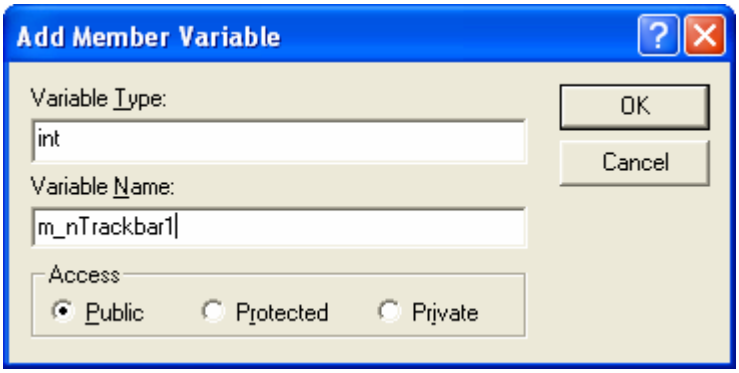


Figure 66: Entering the variable type and name.

```
class CMymfc8Dialog : public CDialog
{
// Construction
public:
    int m_nTrackbar1;
    int m_nProgress;
    CMymfc8Dialog(CWnd* pParent = NULL);

// Dialog Data
```

Listing 15.

```
// CMymfc8Dialog dialog

CMymfc8Dialog::CMymfc8Dialog(CWnd* pParent /*=NULL*/)
    : CDialog(CMymfc8Dialog::IDD, pParent)
{
    m_nProgress = 0;
    m_nTrackbar1 = 0;

    //{{AFX_DATA_INIT(CMymfc8Dialog)
        // NOTE: the ClassWizard will add member initi
    //}}AFX_DATA_INIT
}
```

Listing 16.

Next add the following code in the `OnInitDialog()` member function to set the trackbar's range, to initialize its position from the data member, and to set the neighboring static control to the tracker's current value.

```
CString strText1;
CSliderCtrl* pSlide1 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR1);
pSlide1->SetRange(0, 100);
pSlide1->SetPos(m_nTrackbar1);
strText1.Format("%d", pSlide1->GetPos());
SetDlgItemText(IDC_STATIC_TRACK1, strText1);
```

```
BOOL CMymfc8Dialog::OnInitDialog()
{
    // TODO: Add extra initialization here
    CProgressCtrl* pProg = (CProgressCtrl*) GetDlgItem(IDC_PROGRESS1);
    pProg->SetRange(0, 100);
    pProg->SetPos(m_nProgress);

    CString strText1;
    CSliderCtrl* pSlide1 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR1);
    pSlide1->SetRange(0, 100);
    pSlide1->SetPos(m_nTrackbar1);
    strText1.Format("%d", pSlide1->GetPos());
    SetDlgItemText(IDC_STATIC_TRACK1, strText1);
}
```

Listing 17.

To keep the static control updated, you need to map the `WM_HSCROLL` message that the trackbar sends to the dialog. Here is the code for the handler:

```
void CMymfc8Dialog::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    CSliderCtrl* pSlide = (CSliderCtrl*) pScrollBar;
    CString strText;
    strText.Format("%d", pSlide->GetPos());
    SetDlgItemText(IDC_STATIC_TRACK1, strText);
}
```

```
void CMymfc8Dialog::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default
    CSliderCtrl* pSlide = (CSliderCtrl*) pScrollBar;
    CString strText;
    strText.Format("%d", pSlide->GetPos());
    SetDlgItemText(IDC_STATIC_TRACK1, strText);
}
```

Listing 18.

Finally, you need to update the trackbar's `m_nTrackbar1` data member when the user clicks **OK**. Your natural instinct would be to put this code in the `OnOK()` button handler. You would have a problem, however, if a data exchange validation error occurred involving any other control in the dialog. Your handler would set `m_nTrackbar1` even though the user might choose to cancel the dialog. To avoid this problem, add your code in the `DoDataExchange()` function as shown below. If you do your own validation and detect a problem, call the `CDataExchange::Fail` function, which alerts the user with a message box.

```
    if (pDX->m_bSaveAndValidate)
    {
        TRACE("updating trackbar data members\n");
        CSliderCtrl* pSlide1 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR1);
        m_nTrackbar1 = pSlide1->GetPos();

    }
```

```
void CMymfc8Dialog::DoDataExchange(CDataExchange* pDX)
{
    //{{AFX_DATA_MAP(CMymfc8Dialog)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
 if (pDX->m_bSaveAndValidate)
{
    TRACE("updating trackbar data members\n");
    CSliderCtrl* pSlide1 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR1);
    m_nTrackbar1 = pSlide1->GetPos();

}
}
```

Listing 19.

Program the "discrete" trackbar control. Add a public integer data member named `m_nTrackbar2` to the `CMymfc8Dialog` header, and set it to **0** in the constructor. This data member is a zero-based index into the `dValue`, the array of numbers (4.0, 5.6, 8.0, 11.0, and 16.0) that the trackbar can represent.

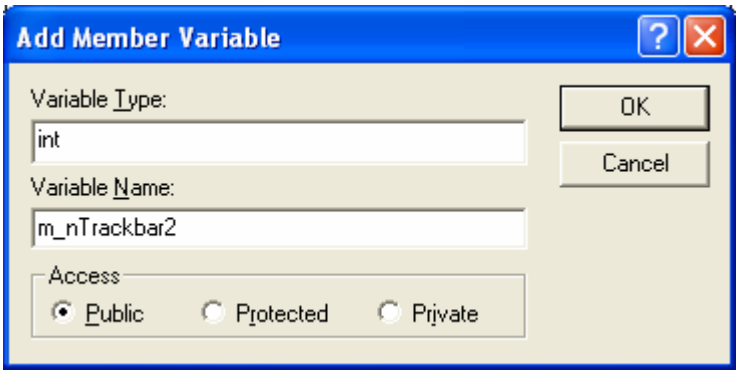


Figure 67: Adding a public integer data member named `m_nTrackbar2` to the `CMymfc8Dialog` header.

```
class CMymfc8Dialog : public CDialog
{
// Construction
public:
    int m_nTrackbar2;
    int m_nTrackbar1;
    int m_nProgress;
    CMymfc8Dialog(CWnd* pParent = NULL);
```

Listing 20.

```
// CMymfc8Dialog dialog

CMymfc8Dialog::CMymfc8Dialog(CWnd* pParent /*=NULL*/)
    : CDialog(CMymfc8Dialog::IDD, pParent)
{
    m_nProgress = 0;
    m_nTrackbar1 = 0;
    m_nTrackbar2 = 0;

    //{{AFX_DATA_INIT(CMymfc8Dialog)
        // NOTE: the ClassWizard will add member initi
```

Listing 21.

Define `dValue` as a private static double array member variable in **mymfc8Dialog.h**:

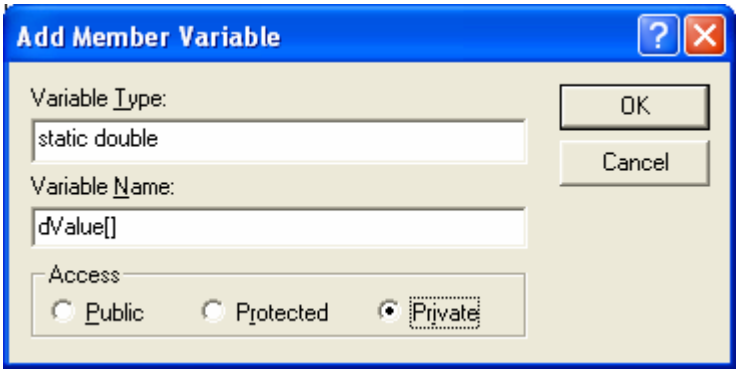


Figure 68: Adding `dValue` array member variable in **mymfc8Dialog.h**.

```
    afx_msg void OnVScroll(UINT
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    static double dValue[];
};
```

Listing 22.

And add to **mymfc8Dialog.cpp** the following line:

```
double CMymfc8Dialog::dValue[5] = {4.0, 5.6, 8.0, 11.0, 16.0};
```

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

/////////////////////////////////////////////////////////////////////////////
// CMymfc8Dialog dialog

double CMymfc8Dialog::dValue[5] = {4.0, 5.6, 8.0, 11.0, 16.0};

CMymfc8Dialog::CMymfc8Dialog(CWnd* pParent /*=NULL*/)
```

Listing 23.

Next add code in the `OnInitDialog()` member function to set the trackbar's range and initial position.

```
CString strText2;
```

```
        CSliderCtrl* pSlide2 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR2);
        pSlide2->SetRange(0, 4);
        pSlide2->SetPos(m_nTrackbar2);
        strText2.Format("%3.1f", dValue[pSlide2->GetPos()]);
        SetDlgItemText(IDC_STATIC_TRACK2, strText2);
```

```
BOOL CMymfc8Dialog::OnInitDialog()
{
    // TODO: Add extra initialization here
    CProgressCtrl* pProg = (CProgressCtrl*) GetDlgItem(IDC_PROGRESS1);
    pProg->SetRange(0, 100);
    pProg->SetPos(m_nProgress);

    CString strText1;
    CSliderCtrl* pSlide1 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR1);
    pSlide1->SetRange(0, 100);
    pSlide1->SetPos(m_nTrackbar1);
    strText1.Format("%d", pSlide1->GetPos());
    SetDlgItemText(IDC_STATIC_TRACK1, strText1);

    CString strText2;
    CSliderCtrl* pSlide2 =
    (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR2);
    pSlide2->SetRange(0, 4);
    pSlide2->SetPos(m_nTrackbar2);
    strText2.Format("%3.1f", dValue[pSlide2->GetPos()]);
    SetDlgItemText(IDC_STATIC_TRACK2, strText2);
}
```

Listing 24.

If you had only one trackbar, the `WM_HSCROLL` handler in the previous step would work. But because you have two trackbars that send `WM_HSCROLL` messages, the handler must differentiate. Here is the new code:

```
        void CMymfc8Dialog::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
        {
            CSliderCtrl* pSlide = (CSliderCtrl*) pScrollBar;
            CString strText;

            // Two trackbars are sending
            // HSCROLL messages (different processing)
            switch(pScrollBar->GetDlgCtrlID())
           {
            case IDC_TRACKBAR1:
                strText.Format("%d", pSlide->GetPos());
                SetDlgItemText(IDC_STATIC_TRACK1, strText);
                break;
            case IDC_TRACKBAR2:
                strText.Format("%3.1f", dValue[pSlide->GetPos()]);
                SetDlgItemText(IDC_STATIC_TRACK2, strText);
                break;
            }
        }
```

```
void CMymfc8Dialog::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default
    CSliderCtrl* pSlide = (CSliderCtrl*) pScrollBar;
    CString strText;

    // Two trackbars are sending
    //  HSCROLL messages (different processing)
    switch(pScrollBar->GetDlgCtrlID())
   {
    case IDC_TRACKBAR1:
        strText.Format("%d", pSlide->GetPos());
        SetDlgItemText(IDC_STATIC_TRACK1, strText);
        break;
    case IDC_TRACKBAR2:
        strText.Format("%3.1f", dValue[pSlide->GetPos()]);
        SetDlgItemText(IDC_STATIC_TRACK2, strText);
        break;
    }
}
```

Listing 25.

This trackbar needs tick marks, so you must check the control's **Tick Marks** and **Auto Ticks** properties back in the dialog editor. With **Auto Ticks** set, the trackbar will place a tick at every increment. The same data exchange considerations applied to the previous trackbar applies to this trackbar. Add the following code in the dialog class `DoDataExchange()` member function inside the block for the `if` statement you added in the previous step:

```
CSliderCtrl* pSlide2 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR2);
m_nTrackbar2 = pSlide2->GetPos();
```

```
void CMymfc8Dialog::DoDataExchange(CDataExchange* pDX)
{
    //{{AFX_DATA_MAP(CMymfc8Dialog)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
 if (pDX->m_bSaveAndValidate)
{
    TRACE("updating trackbar data members\n");
    CSliderCtrl* pSlide1 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR1);
    m_nTrackbar1 = pSlide1->GetPos();

    CSliderCtrl* pSlide2 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR2);
    m_nTrackbar2 = pSlide2->GetPos();
}
}
```

Listing 26.

Use the dialog editor to set the **Point** property of both trackbars to **Bottom/Right**. Select **Right** for the **Align Text** property of both the `IDC_STATIC_TRACK1` and `IDC_STATIC_TRACK2` static controls.

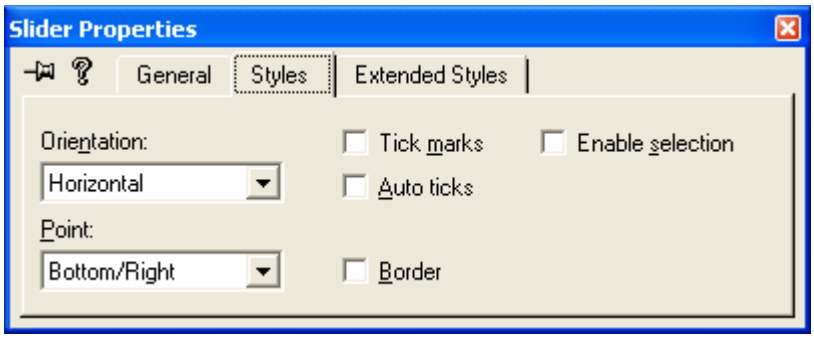


Figure 69: Modifying the **Slider** properties.

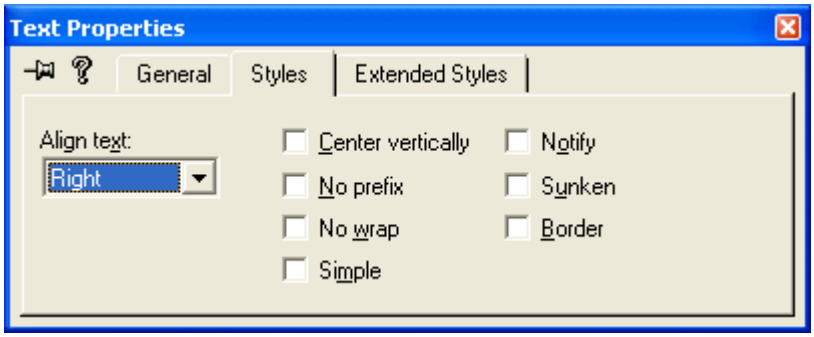


Figure 70: Modifying **static text** properties.

Program the spin button control. The spin control depends on its buddy edit control, located immediately before it in the tab order. Use ClassWizard to add a `double`-precision data member called `m_dSpin` for the `IDC_BUDDY_SPIN1` edit control.

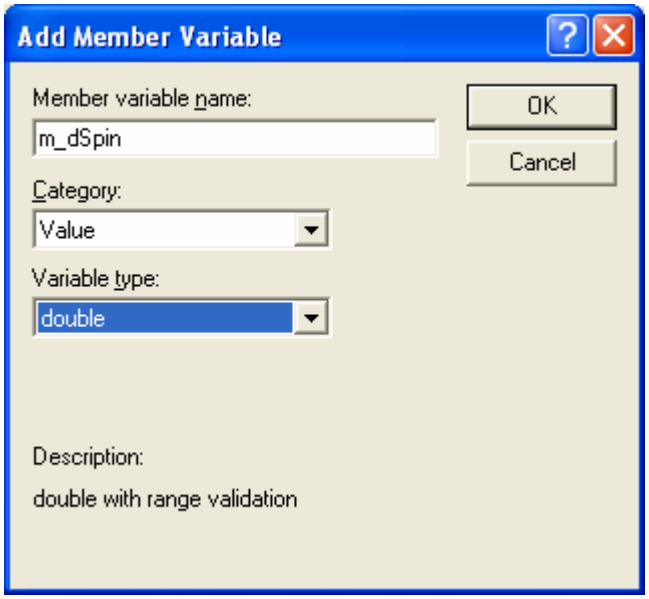


Figure 71: Adding a `double`-precision `m_dSpin` for the `IDC_BUDDY_SPIN1` edit control.

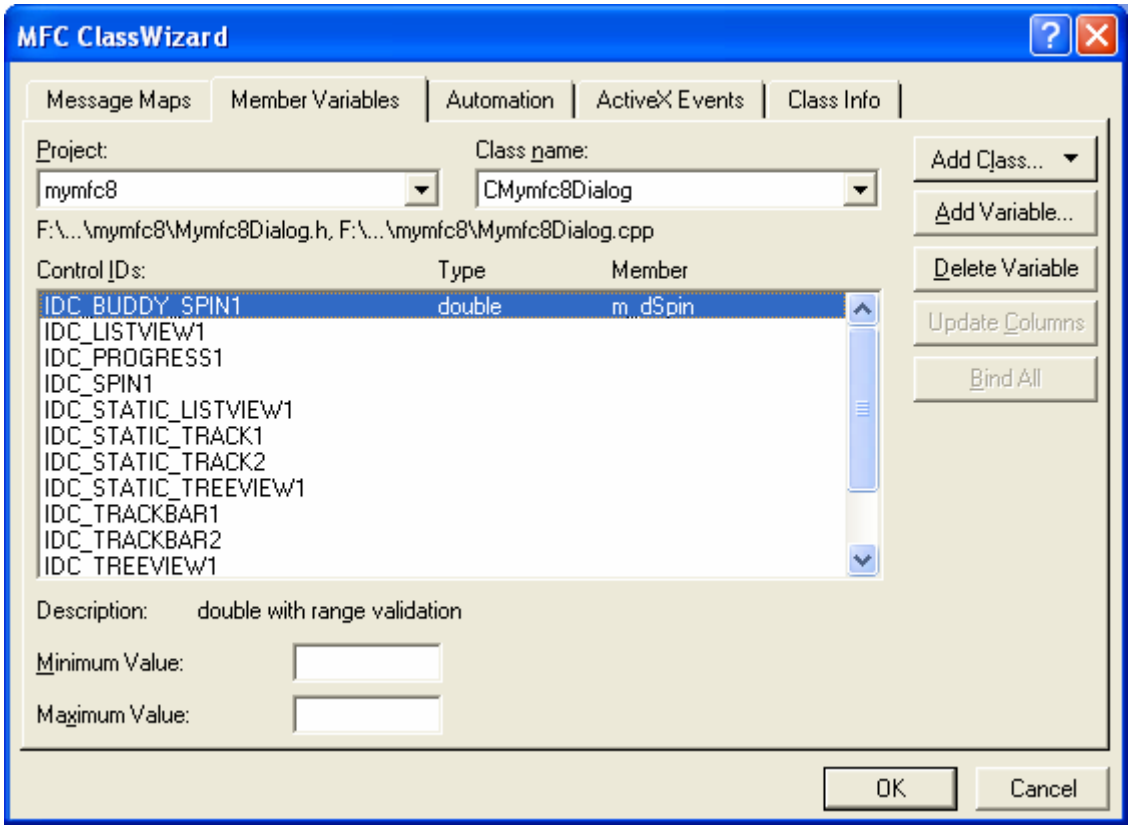


Figure 72: The added member variable.

We're using a `double` instead of an `int` because the int would require almost no programming, and that would be too easy. We want the edit control range to be 0.0 to 10.0, but the spin control itself needs an integer range. Add the following code to `OnInitDialog()` to set the spin control range to 0 to 100 and to set its initial value to `m_dSpin * 10.0`:

```
CSpinButtonCtrl* pSpin = (CSpinButtonCtrl*) GetDlgItem(IDC_SPIN1);
pSpin->SetRange(0, 100);
pSpin->SetPos((int) (m_dSpin * 10.0));
```

```
BOOL CMymfc8Dialog::OnInitDialog()
{
    // TODO: Add extra initialization here
    CProgressCtrl* pProg = (CProgressCtrl*) GetDlgItem(IDC_PROGRESS1);
    pProg->SetRange(0, 100);
    pProg->SetPos(m_nProgress);

    CString strText1;
    CSliderCtrl* pSlide1 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR1);
    pSlide1->SetRange(0, 100);
    pSlide1->SetPos(m_nTrackbar1);
    strText1.Format("%d", pSlide1->GetPos());
    SetDlgItemText(IDC_STATIC_TRACK1, strText1);

    CString strText2;
    CSliderCtrl* pSlide2 = (CSliderCtrl*) GetDlgItem(IDC_TRACKBAR2);
    pSlide2->SetRange(0, 4);
    pSlide2->SetPos(m_nTrackbar2);
    strText2.Format("%3.1f", dValue[pSlide2->GetPos()]);
    SetDlgItemText(IDC_STATIC_TRACK2, strText2);

    CSpinButtonCtrl* pSpin = (CSpinButtonCtrl*) GetDlgItem(IDC_SPIN1);
    pSpin->SetRange(0, 100);
    pSpin->SetPos((int) (m_dSpin * 10.0));
}
```

Listing 27.

To display the current value in the buddy edit control, you need to map the `WM_VSCROLL` message that the spin control sends to the dialog. Here's the code:

```
void CMymfc8Dialog::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    if (nSBCode == SB_ENDSCROLL)
    {
        return; // Reject spurious messages
    }
    // Process scroll messages from IDC_SPIN1 only
    if (pScrollBar->GetDlgCtrlID() == IDC_SPIN1)
    {
        CString strValue;
        strValue.Format("%3.1f", (double) nPos / 10.0);
        ((CSpinButtonCtrl*) pScrollBar)->GetBuddy()->SetWindowText(strValue);
    }
}
```

```
void CMymfc8Dialog::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default
    if (nSBCode == SB_ENDSCROLL)
    {
        return; // Reject spurious messages
    }
    // Process scroll messages from IDC_SPIN1 only
    if (pScrollBar->GetDlgCtrlID() == IDC_SPIN1)
    {
        CString strValue;
        strValue.Format("%3.1f", (double) nPos / 10.0);
        ((CSpinButtonCtrl*) pScrollBar)->GetBuddy()->SetWindowText(strValue);
    }
}
```

Listing 28.

There's no need for you to add code in `OnOK()` or in `DoDataExchange()` because the dialog data exchange code processes the contents of the edit control. In the dialog editor, select the spin control's **Auto Buddy** property and the buddy's **Read-only** property.

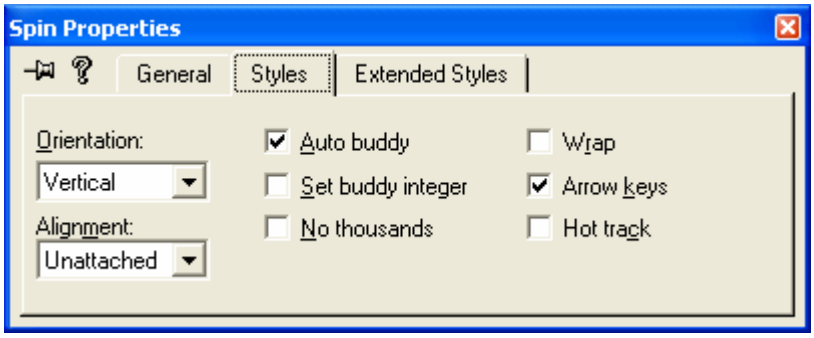


Figure 73: Modifying the **Spin** control properties.

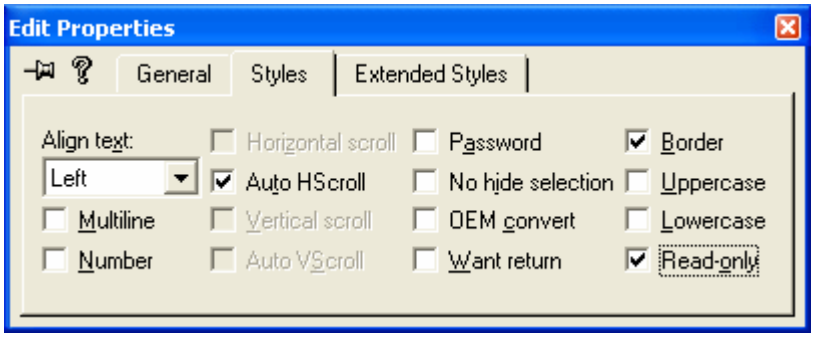


Figure 74: Modifying the **Edit** control properties.

Set up an image list. Both the **list control** and the **tree control** need an image list, and the image list needs icons. First use the graphics editor to add icons to the project's RC file.

Figure 75: Inserting new icons.

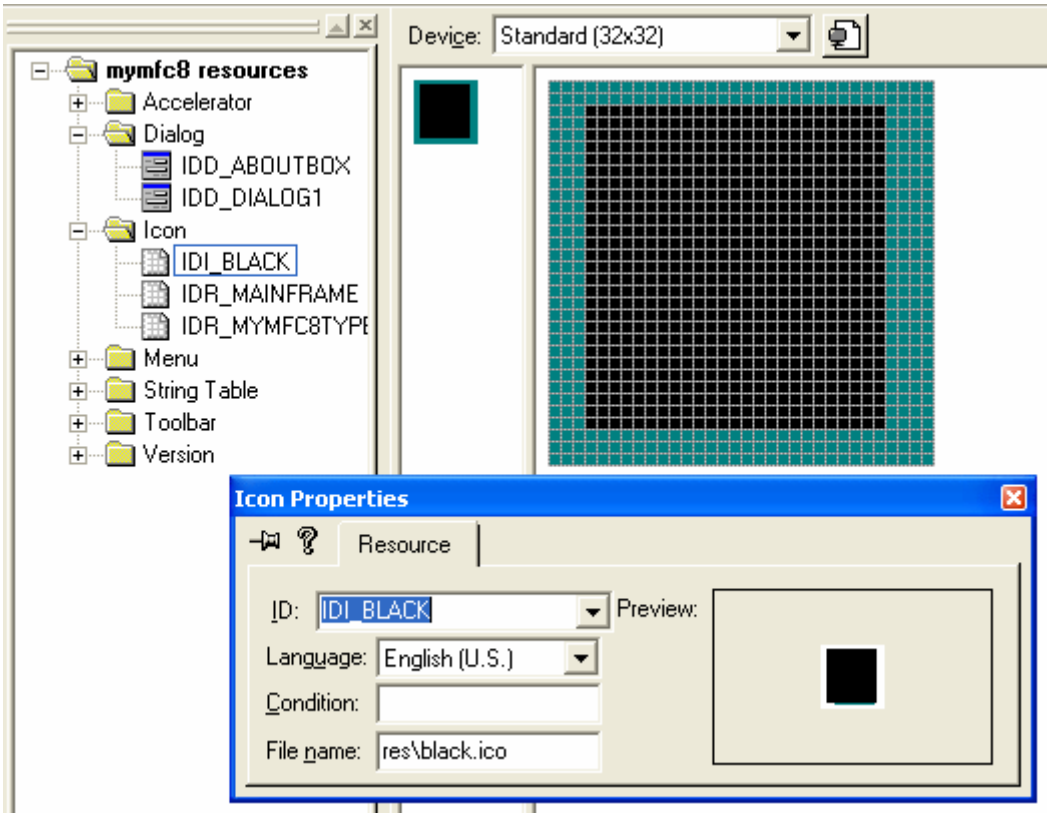


Figure 76: Modifying icon properties in resource editor.

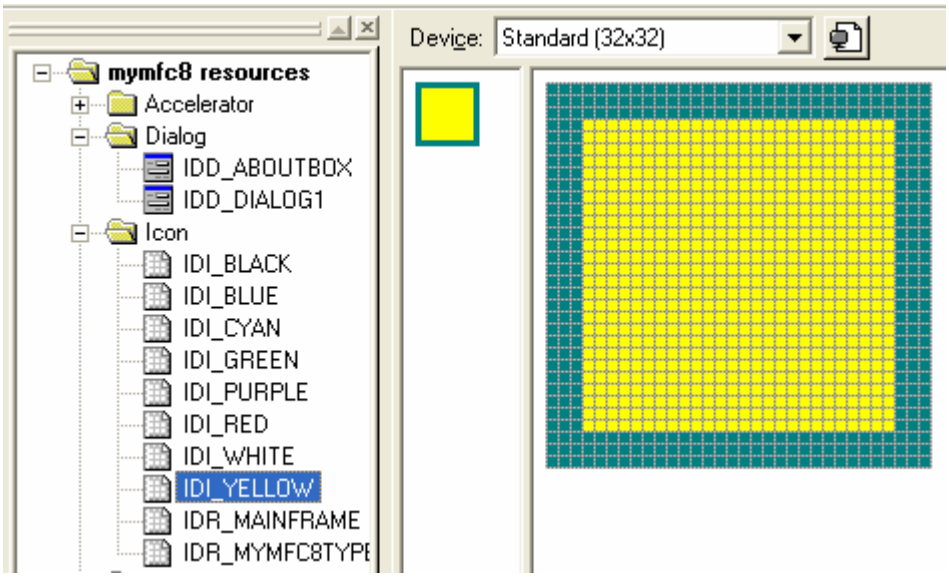


Figure 77: Completed icons creation.

Use fancier icons if you have them. You can import an icon by choosing **Resource** from the **Insert** menu and then clicking the **Import** button. For this example, the icon resource IDs are as follows.

Resource ID	Icon File name
IDI_BLACK	Icon1
IDI_BLUE	Icon3

IDI_CYAN	Icon5
IDI_GREEN	Icon7
IDI_PURPLE	Icon6
IDI_RED	Icon2
IDI_WHITE	Icon0
IDI_YELLOW	Icon4

Table 3: Icons resource IDs.

Next add a private `CImageList` data member called `m_imageList` in the `CMymfc8Dialog` class header.

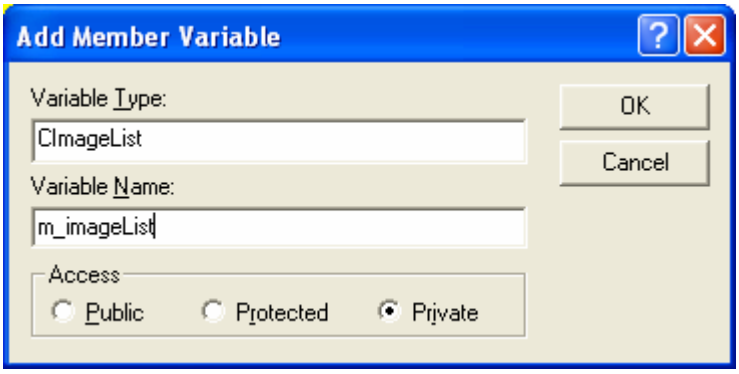


Figure 78: Adding a private `CImageList` data member, `m_imageList` in the `CMymfc8Dialog` class header.

```
    afx_msg void OnVScroll(U
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CImageList m_imageList;
    static double dValue[];
};
```

Listing 29.

And then add the following code to `OnInitDialog()`:

```
HICON hIcon[8];
int n;
m_imageList.Create(16, 16, 0, 8, 8); // 32, 32 for large icons
hIcon[0] = AfxGetApp()->LoadIcon(IDI_WHITE);
hIcon[1] = AfxGetApp()->LoadIcon(IDI_BLACK);
hIcon[2] = AfxGetApp()->LoadIcon(IDI_RED);
hIcon[3] = AfxGetApp()->LoadIcon(IDI_BLUE);
hIcon[4] = AfxGetApp()->LoadIcon(IDI_YELLOW);
hIcon[5] = AfxGetApp()->LoadIcon(IDI_CYAN);
hIcon[6] = AfxGetApp()->LoadIcon(IDI_PURPLE);
hIcon[7] = AfxGetApp()->LoadIcon(IDI_GREEN);
for (n = 0; n < 8; n++)
{
    m_imageList.Add(hIcon[n]);
}
```

```
    CSpinButtonCtrl* pSpin = (CSpinButtonCtrl*) GetDlgItem(IDC_SPIN1);
    pSpin->SetRange(0, 100);
    pSpin->SetPos((int) (m_dSpin * 10.0));
    HICON hIcon[8];
    int n;
    m_imageList.Create(16, 16, 0, 8, 8); // 32, 32 for large icons
    hIcon[0] = AfxGetApp()->LoadIcon(IDI_WHITE);
    hIcon[1] = AfxGetApp()->LoadIcon(IDI_BLACK);
    hIcon[2] = AfxGetApp()->LoadIcon(IDI_RED);
    hIcon[3] = AfxGetApp()->LoadIcon(IDI_BLUE);
    hIcon[4] = AfxGetApp()->LoadIcon(IDI_YELLOW);
    hIcon[5] = AfxGetApp()->LoadIcon(IDI_CYAN);
    hIcon[6] = AfxGetApp()->LoadIcon(IDI_PURPLE);
    hIcon[7] = AfxGetApp()->LoadIcon(IDI_GREEN);
    for (n = 0; n < 8; n++)
    {
        m_imageList.Add(hIcon[n]);
    }
}
```

CSpinButtonCtrl *pSpin

Listing 30.

About Icons

You probably know that a bitmap is an array of bits that represent pixels on the display. In Windows, an icon is a "bundle" of bitmaps. First of all, an icon has different bitmaps for different sizes. Typically, small icons are 16-by-16 pixels and large icons are 32-by-32 pixels. Within each size are two separate bitmaps: one 4-bit-per-pixel bitmap for the color image and one monochrome (1-bit-per-pixel) bitmap for the "mask." If a mask bit is 0, the corresponding image pixel represents an opaque color. If the mask bit is 1, an image color of black (0) means that the pixel is transparent and an image color of white (`0xF`) means that the background color is inverted at the pixel location. Windows 95 and Windows NT seem to process inverted colors a little differently than Windows 3.x does, the inverted pixels show up transparent against the desktop, black against a Windows Explorer window background, and white against list and tree control backgrounds. Don't ask me why.

Small icons were new with Windows 95. They're used in the task bar, in Windows Explorer, and in your list and tree controls, if you want them there. If an icon doesn't have a 16-by-16-pixel bitmap, Windows manufactures a small icon out of the 32-by-32-pixel bitmap, but it won't be as neat as one you draw yourself. The graphics editor lets you create and edit icons. Look at the color palette shown here.

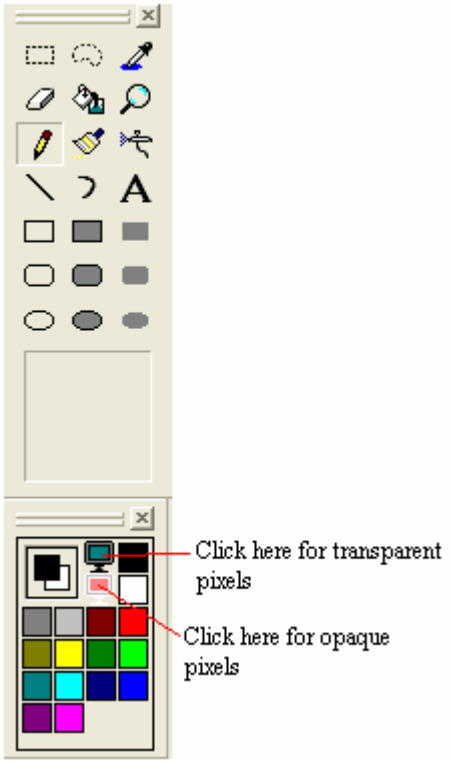


Figure 79: Color palette and other utilities for icon editing.

The top square in the upper-left portion shows you the main color for brushes, shape interiors, and so on, and the square under it shows the border color for shape outlines. You select a main color by left-clicking on a color, and you select a border color by right-clicking on a color. Now look at the top center portion of the color palette. You click on the upper "monitor" to paint transparent pixels, which are drawn in dark cyan. You click on the lower monitor to paint inverted pixels, which are drawn in red.

Program the list control. In the dialog editor, set the list control's style attributes as shown in the next illustration.

Figure 80: Modifying the **list** control properties.

Make sure the **Border** style on the **More Styles** page is set. Next add the following code to `OnInitDialog()`:

```
static char* color[] = {"white", "black", "red",
                        "blue", "yellow", "cyan",
                        "purple", "green"};
CListCtrl* pList = (CListCtrl*) GetDlgItem(IDC_LISTVIEW1);
pList->SetImageList(&m_imageList, LVSIL_SMALL);
for (n = 0; n < 8; n++)
{
    pList->InsertItem(n, color[n], n);
```

```
    }
    pList->SetBkColor(RGB(0, 255, 255)); // UGLY!
    pList->SetTextBkColor(RGB(0, 255, 255));
```

```
    static char* color[] = {"white", "black", "red",
                            "blue", "yellow", "cyan",
                            "purple", "green"};
    CListCtrl* pList = (CListCtrl*) GetDlgItem(IDC_LISTVIEW1);
    pList->SetImageList(&m_imageList, LVSIL_SMALL);
    for (n = 0; n < 8; n++)
    {
        pList->InsertItem(n, color[n], n);
    }
    pList->SetBkColor(RGB(0, 255, 255)); // UGLY!
    pList->SetTextBkColor(RGB(0, 255, 255));
}
```

Listing 31.

As the last two lines illustrate, you don't use the `WM_CTLCOLOR` message with common controls; you just call a function to set the background color. As you'll see when you run the program, however, the icons' inverse-color pixels look shabby. If you use ClassWizard to map the list control's `LVN_ITEMCHANGED` notification message, so that you'll be able to track the user's selection of items.

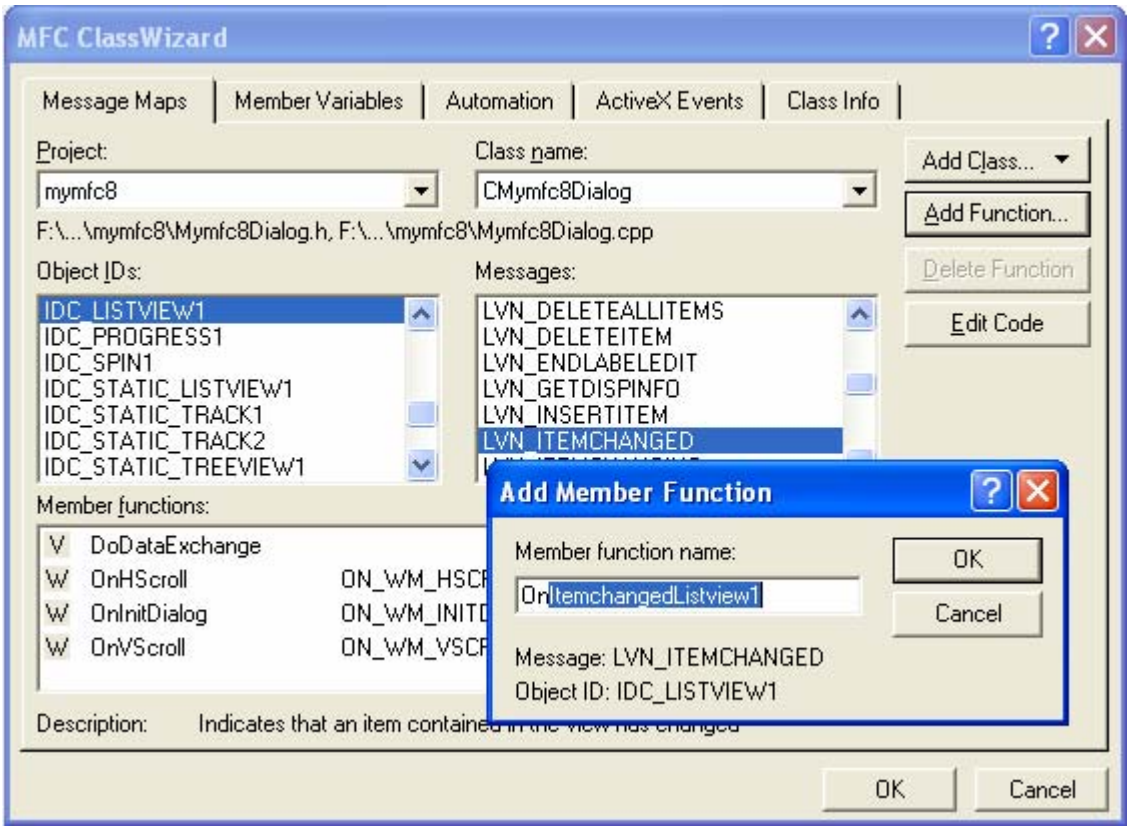


Figure 81: Mapping the list control's `LVN_ITEMCHANGED` notification message.

The code in the following handler displays the selected item's text in a static control:

```
void CMymfc8Dialog::OnItemchangedListview1(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_LISTVIEW* pNMListView = (NM_LISTVIEW*)pNMHDR;
    CListCtrl* pList = (CListCtrl*) GetDlgItem(IDC_LISTVIEW1);
    int nSelected = pNMListView->iItem;
```

```
        if (nSelected >= 0)
        {
            CString strItem = pList->GetItemText(nSelected, 0);
            SetDlgItemText(IDC_STATIC_LISTVIEW1, strItem);
        }
        *pResult = 0;
    }
```

```
void CMymfc8Dialog::OnItemchangedListview1(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_LISTVIEW* pNMListView = (NM_LISTVIEW*)pNMHDR;
    CListCtrl* pList = (CListCtrl*) GetDlgItem(IDC_LISTVIEW1);
    int nSelected = pNMListView->iItem;
    if (nSelected >= 0)
    {
        CString strItem = pList->GetItemText(nSelected, 0);
        SetDlgItemText(IDC_STATIC_LISTVIEW1, strItem);
    }
    *pResult = 0;
}
```

Listing 32.

The `NM_LISTVIEW` structure has a data member called `iItem` that contains the index of the selected item.

Program the tree control. In the dialog editor, set the **tree control**'s style attributes as shown here.

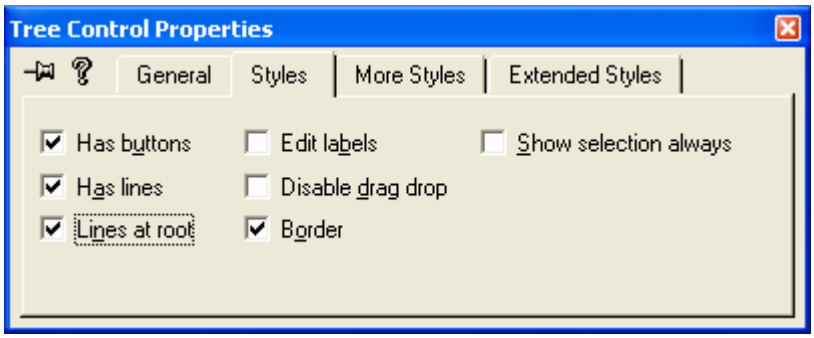


Figure 82: Modifying the **tree** control styles.

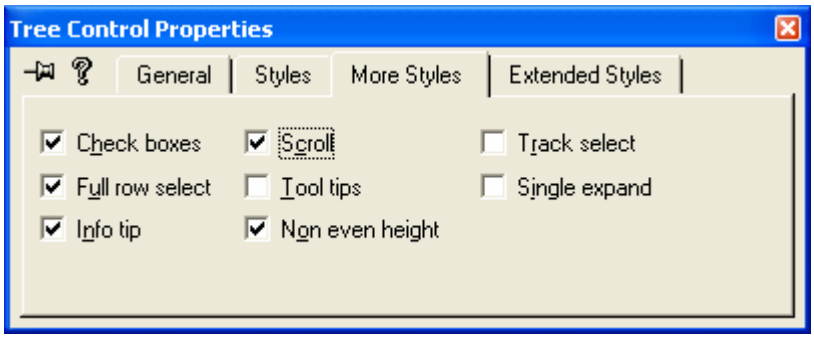


Figure 83: Another modification of the tree control styles.

Next, add the following lines to `OnInitDialog()`:

```
    CTreeCtrl* pTree = (CTreeCtrl*) GetDlgItem(IDC_TREEVIEW1);
    pTree->SetImageList(&m_imageList, TVSIL_NORMAL);
    // tree structure common values
    TV_INSERTSTRUCT tvinsert;
    tvinsert.hParent = NULL;
```

```
tvinsert.hInsertAfter = TVI_LAST;
tvinsert.item.mask = TVIF_IMAGE | TVIF_SELECTEDIMAGE | TVIF_TEXT;
tvinsert.item.hItem = NULL;
tvinsert.item.state = 0;
tvinsert.item.stateMask = 0;
tvinsert.item.cchTextMax = 6;
tvinsert.item.iSelectedImage = 1;
tvinsert.item.cChildren = 0;
tvinsert.item.lParam = 0;
// top level
tvinsert.item.pszText = "Homer";
tvinsert.item.iImage = 2;
HTREEITEM hDad = pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Marge";
HTREEITEM hMom = pTree->InsertItem(&tvinsert);
// second level
tvinsert.hParent = hDad;
tvinsert.item.pszText = "Bart";
tvinsert.item.iImage = 3;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Lisa";
pTree->InsertItem(&tvinsert);
// second level
tvinsert.hParent = hMom;
tvinsert.item.pszText = "Bart";
tvinsert.item.iImage = 4;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Lisa";
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Dilbert";
HTREEITEM hOther = pTree->InsertItem(&tvinsert);
// third level
tvinsert.hParent = hOther;
tvinsert.item.pszText = "Dogbert";
tvinsert.item.iImage = 7;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Ratbert";
pTree->InsertItem(&tvinsert);
```

```
CTreeCtrl* pTree = (CTreeCtrl*) GetDlgItem(IDC_TREEVIEW1);
pTree->SetImageList(&m_imageList, TVSIL_NORMAL);
// tree structure common values
TV_INSERTSTRUCT tvinsert;
tvinsert.hParent = NULL;
tvinsert.hInsertAfter = TVI_LAST;
tvinsert.item.mask = TVIF_IMAGE | TVIF_SELECTEDIMAGE |
                     TVIF_TEXT;
tvinsert.item.hItem = NULL;
tvinsert.item.state = 0;
tvinsert.item.stateMask = 0;
tvinsert.item.cchTextMax = 6;
tvinsert.item.iSelectedImage = 1;
tvinsert.item.cChildren = 0;
tvinsert.item.lParam = 0;
// top level
tvinsert.item.pszText = "Homer";
tvinsert.item.iImage = 2;
HTREEITEM hDad = pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Marge";
HTREEITEM hMom = pTree->InsertItem(&tvinsert);
// second level
tvinsert.hParent = hDad;
tvinsert.item.pszText = "Bart";
tvinsert.item.iImage = 3;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Lisa";
pTree->InsertItem(&tvinsert);
// second level
tvinsert.hParent = hMom;
tvinsert.item.pszText = "Bart";
tvinsert.item.iImage = 4;
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Lisa";
pTree->InsertItem(&tvinsert);
tvinsert.item.pszText = "Dilbert";
HTREEITEM hOther = pTree->InsertItem(&tvinsert);
// third level
tvinsert.hParent = hOther;
tvinsert.item.pszText = "Dogbert";
```

Listing 33.

As you can see, this code sets `TV_INSERTSTRUCT` text and image indexes and calls `InsertItem()` to add nodes to the tree. Finally, use ClassWizard to map the `TVN_SELCHANGED` notification for the tree control.

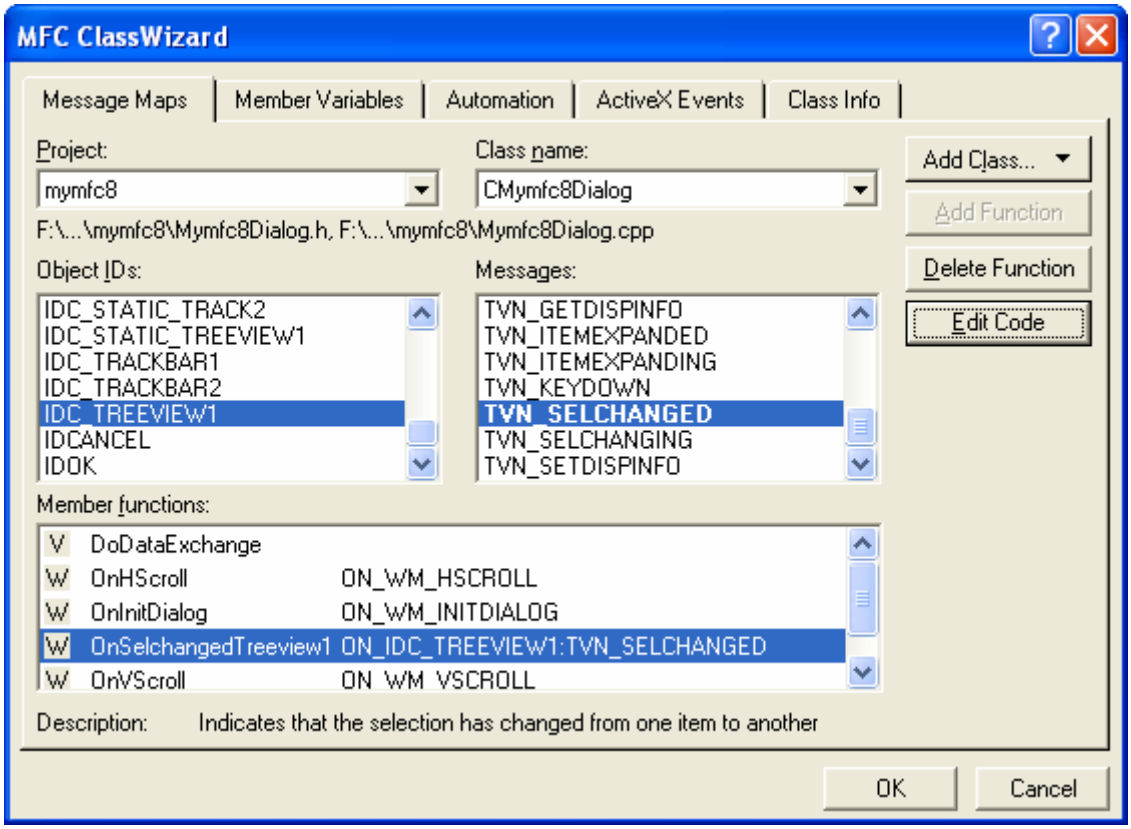


Figure 84: Mapping the TVN_SELCHANGED notification for the tree control.

Click the **Edit Code** button and add the handler code to display the selected text in a static control:

```
void CMymfc8Dialog::OnSelchangedTreeview1(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;
    CTreeCtrl* pTree = (CTreeCtrl*) GetDlgItem(IDC_TREEVIEW1);
    HTREEITEM hSelected = pNMTreeView->itemNew.hItem;
    if (hSelected != NULL)
    {
        char text[31];
        TV_ITEM item;
        item.mask = TVIF_HANDLE | TVIF_TEXT;
        item.hItem = hSelected;
        item.pszText = text;
        item.cchTextMax = 30;
        VERIFY(pTree->GetItem(&item));
        SetDlgItemText(IDC_STATIC_TREEVIEW1, text);
    }
    *pResult = 0;
}
```

```
void CMymfc8Dialog::OnSelchangedTreeview1(NMHDR* pNMHDR, LRESULT* pResult)
{
    // TODO: Add your control notification handler code here
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;
    CTreeCtrl* pTree = (CTreeCtrl*) GetDlgItem(IDC_TREEVIEW1);
    HTREEITEM hSelected = pNMTreeView->itemNew.hItem;
    if (hSelected != NULL) {
        char text[31];
        TV_ITEM item;
        item.mask = TVIF_HANDLE | TVIF_TEXT;
        item.hItem = hSelected;
        item.pszText = text;
        item.cchTextMax = 30;
        VERIFY(pTree->GetItem(&item));
        SetDlgItemText(IDC_STATIC_TREEVIEW1, text);
    }
    *pResult = 0;
}
```

Listing 34.

The `NM_TREEVIEW` structure has a data member called `itemNew` that contains information about the selected node; `itemNew.hItem` is the handle of that node. The `GetItem()` function retrieves the node's data, storing the text using a pointer supplied in the `TV_ITEM` structure. The mask variable tells Windows that the `hItem` handle is valid going in and that text output is desired.

Add code to the virtual `OnDraw()` function in file **mymfc8View.cpp**. The following code replaces the previous code:

```
void CMymfc8View::OnDraw(CDC* pDC)
{
    pDC->TextOut(30, 30, "Press the left mouse button here.");
}
```

```
void CMymfc8View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    pDC->TextOut(30, 30, "Press the left mouse button here.");
}
```

Listing 35.

Use ClassWizard to add the `OnLButtonDown()` member function.

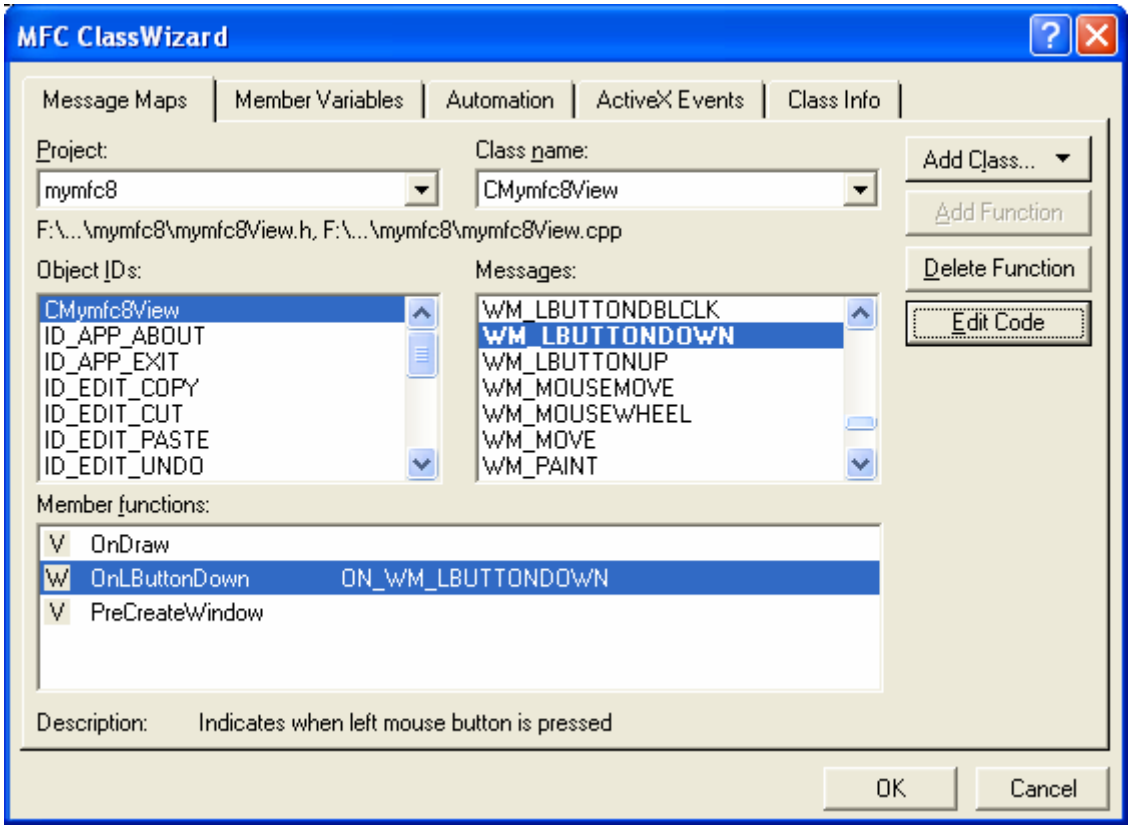


Figure 85: Adding the `OnLButtonDown()` member function to handle the left mouse click event.

Edit the AppWizard-generated code as follows:

```
void CMymfc8View::OnLButtonDown(UINT nFlags, CPoint point)
{
    CMymfc8Dialog dlg;

    dlg.m_nTrackbar1 = 20;
    dlg.m_nTrackbar2 = 2; // index for 8.0
    dlg.m_nProgress = 70; // write-only
    dlg.m_dSpin = 3.2;

    dlg.DoModal();
}
```

```
void CMymfc8View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CMymfc8Dialog dlg;

    dlg.m_nTrackbar1 = 20;
    dlg.m_nTrackbar2 = 2; // index for 8.0
    dlg.m_nProgress = 70; // write-only
    dlg.m_dSpin = 3.2;

    dlg.DoModal();
}
```

Listing 36.

Add a statement to include **mymfc8Dialog.h** in file **mymfc8View.cpp**.

```
// mymfc8View.cpp : implementation of the CMymfc8View class

#include "stdafx.h"
#include "mymfc8.h"

#include "mymfc8Doc.h"
#include "mymfc8View.h"
#include "mymfc8Dialog.h"

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

If you unintentionally deleted the return value of the very long code for `CMymfc8Dialog::OnInitDialog`, make sure it is like the following code.

```
      return CDialog::OnInitDialog();
```

```
    pTree->InsertItem(&tvinsert);
    tvinsert.item.pszText = "Ratbert";
    pTree->InsertItem(&tvinsert);
    // Call after initialization
    return CDialog::OnInitDialog();
}
```

Listing 37.

Finally, build and run the program. Experiment with the controls to see how they work. We haven't added code to make the progress indicator functional that will be covered in Module 22.

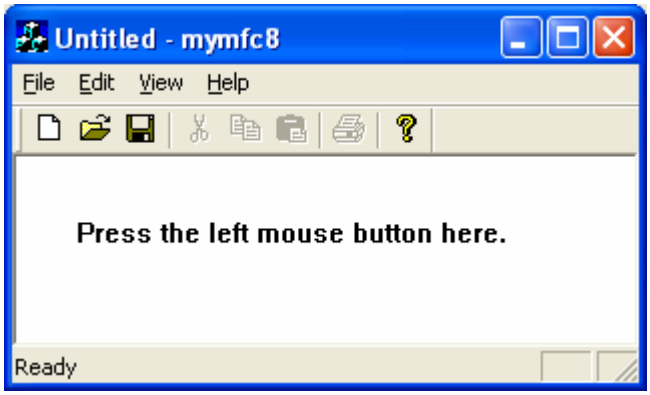


Figure 86: The MYMFC8 program output.

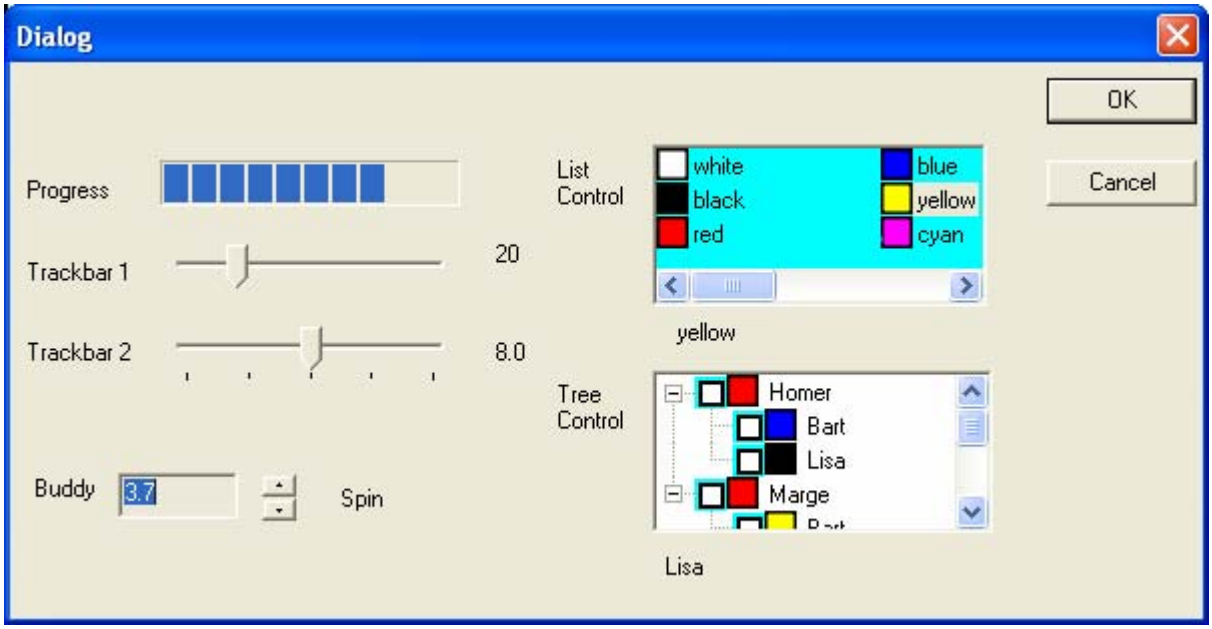


Figure 87: MYMFC8 program output when the left mouse button is clicked, dialog with full of the common controls.

Other Windows Common Controls

You've seen most of the common controls that appear on the dialog editor control palette. We've skipped the **animation control** because this book doesn't cover multimedia, and we've skipped the **hot key control** because it isn't very interesting. The tab control is interesting, but you seldom use it inside another dialog. Module 7 shows you how to construct a **tabbed dialog**, sometimes known as a property sheet. In Module 7, you'll also see an application that is built around the `CRichEditView` class, which incorporates the Windows rich edit control.

Further reading and digging:

1. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
2. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
3. MSDN Library
4. Windows data type.
5. Win32 programming Tutorial.
6. The best of C/C++, MFC, Windows and other related books.
7. Unicode and Multibyte character set: Story and program examples.