Module 10: Separating the Document from Its View

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:

Separating the Document from Its View
Document - View Interaction Functions
The `CView::GetDocument` Function
The `CDocument::UpdateAllViews` Function
The `CView::OnUpdate` Function
The `CView::OnInitialUpdate` Function
The `CDocument::OnNewDocument` Function
The Simplest Document - View Application
The `CFormView` Class
The `CObject` Class
Diagnostic Dumping
The `TRACE` Macro
The `afxDump` Object
The Dump Context and the `CObject` Class
Automatic Dump of Undeleted Objects
Window Subclassing for Enhanced Data-Entry Control
The MYMFC15 Example
A More Advanced Document-View Interaction
The `CDocument::DeleteContents` Function
The `CObList` Collection Class
Using the `CObList` Class for a First-In, First-Out List
`CObList` Iteration: The `POSITION` Variable
The `CTypedPtrList` Template Collection Class
The Dump Context and Collection Classes
The MYMFC16 Example
Resource Requirements
Toolbar
Student Menu
Edit Menu
The `IDD_MYMFC16_FORM` Dialog Template
Code Requirements
`CMymfc16App` Class
`CMainFrame` Class
`CStudent` Class
ClassWizard and `CMymfc16Doc` Class
Data Members
Constructor
`GetList()`
`DeleteContents()`
`Dump()`
`CMymfc16Doc` Class
ClassWizard and `CMymfc16View`
Data Members
`OnInitialUpdate()`
`OnUpdate()`
Protected Virtual Functions
`CMymfc16View` Class
Testing the MYMFC16 Application

Two Exercises for the Reader

Separating the Document from Its View

Now you're finally going to see the interaction between documents and views. Module 13 gave you a preview of this interaction when it showed the routing of command messages to both view objects and document objects. In this module, you'll see how the document maintains the application's data and how the view presents the data to the user. You'll also learn how the document and view objects talk to each other while the application executes.
The two examples in this module both use the `CFormView` class as the base class for their views. The first example is as simple as possible, with the document holding only one simple object of class `CStudent`, which represents a single student record. The view shows the student's name and grade and allows editing. With the `CStudent` class, you'll get some practice writing classes to represent real-world entities. You'll also get to use the MFC Library version 6.0 diagnostic dump functions. The second example goes further by introducing pointer collection classes, the `CObList` and `CTypedPtrList` classes in particular. Now the document holds a collection of student records, and the view allows the sequencing, insertion, and deletion of individual records.

Document - View Interaction Functions

You already know that the **document object holds the data** and that the **view object displays the data and allows editing**. An SDI application has a document class derived from `CDocument`, and it has one or more view classes, each ultimately derived from `CView`. A complex handshaking process takes place among the document, the view, and the rest of the application framework. To understand this process, you need to know about five important member functions in the document and view classes. Two are non-virtual base class functions that you call in your derived classes; three are virtual functions that you often override in your derived classes. Let's look at these functions one at a time.

The `CView::GetDocument` Function

A view object has one and only one associated document object. The `GetDocument()` function allows an application to navigate from a view to its document. Suppose a view object gets a message that the user has entered new data into an edit control. The view must tell the document object to update its internal data accordingly. The `GetDocument()` function provides the document pointer that can be used to access document class member functions or public data embers.
The `CDocument::GetNextView` function navigates from the document to the view, but because a document can have more than one view, it's necessary to call this member function once for each view, inside a loop. You'll seldom call `GetNextView()` because the application framework provides a better method of iterating through a document's views.
When AppWizard generates a derived `CView` class, it creates a special type-safe version of the `GetDocument()` function that returns not a `CDocument()` pointer but a pointer to an object of your derived class. This function is an inline function, and it looks something like this:

```
CMyDoc* GetDocument()
{
    return (CMyDoc*)m_pDocument;
}
```

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

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

Calls the base class `GetDocument()` function and thus returns a pointer to a `CDocument` object.

The `CDocument::UpdateAllViews` Function

If the document data changes for any reason, all views must be notified so that they can update their representations of that data. If `UpdateAllViews()` is called from a member function of a derived document class, its first parameter,

pSender, is NULL. If UpdateAllViews() is called from a member function of a derived view class, set the pSender parameter to the current view, like this:

```
GetDocument()->UpdateAllViews(this);
```

The non-null parameter prevents the application framework from notifying the current view. The assumption here is that the current view has already updated itself. The function has optional hint parameters that can be used to give view-specific and application-dependent information about which parts of the view to update. This is an advanced use of the function. How exactly is a view notified when UpdateAllViews() gets called? Take a look at the next function, OnUpdate().

The CView::OnUpdate Function

This virtual function is called by the application framework in response to your application's call to the CDocument::UpdateAllViews function. You can, of course, call it directly within your derived CView class. Typically, your derived view class's OnUpdate() function accesses the document, gets the document's data, and then updates the view's data members or controls to reflect the changes. Alternatively, OnUpdate() can invalidate a portion of the view, causing the view's OnDraw() function to use document data to draw in the window. The OnUpdate() function might look something like this:

```
void CMyView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    CMyDocument* pMyDoc = GetDocument();
    CString lastName = pMyDoc->GetLastName();
    // m_pNameStatic is a CMyView data member
    m_pNameStatic->SetWindowText(lastName);
}
```

The hint information is passed through directly from the call to UpdateAllViews(). The default OnUpdate() implementation invalidates the entire window rectangle. In your overridden version, you can choose to define a smaller invalid rectangle as specified by the hint information. If the CDocument() function UpdateAllViews() is called with the pSender parameter pointing to a specific view object, OnUpdate() is called for all the document's views except the specified view.

The CView::OnInitialUpdate Function

This virtual CView function is called when the application starts, when the user chooses **New** from the **File** menu, and when the user chooses **Open** from the **File** menu. The CView base class version of OnInitialUpdate() does nothing but call OnUpdate(). If you override OnInitialUpdate() in your derived view class, be sure that the view class calls the base class's OnInitialUpdate() function or the derived class's OnUpdate() function. You can use your derived class's OnInitialUpdate() function to initialize your view object. When the application starts, the application framework calls OnInitialUpdate() immediately after OnCreate() (if you've mapped OnCreate() in your view class). OnCreate() is called once, but OnInitialUpdate() can be called many times.

The CDocument::OnNewDocument Function

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

The Simplest Document - View Application

Suppose you don't need multiple views of your document but you plan to take advantage of the application framework's file support. In this case, you can forget about the UpdateAllViews() and OnUpdate() functions. Simply follow these steps when you develop the application:

1. In your derived document class header file (generated by AppWizard), declare your document's data members. These data members are the primary data storage for your application. You can make these data members public, or you can declare the derived view class a friend of the document class.
2. In your derived view class, override the `OnInitialUpdate()` virtual member function. The application framework calls this function after the document data has been initialized or read from disk. (Module 11 discusses disk file I/O.) `OnInitialUpdate()` should update the view to reflect the current document data.
3. In your derived view class, let your window message handlers, command message handlers and your `OnDraw()` function read and update the document data members directly, using `GetDocument()` to access the document object.

The sequence of events for this simplified document-view environment is as follows.

Sequence	Description
Application starts	`CMyDocument` object constructed `CMyView` object constructed `View` window created `CMyView::OnCreate` called (if mapped) `CMyDocument::OnNewDocument` called `CMyView::OnInitialUpdate` called `View` object initialized `View` window invalidated `CMyView::OnDraw` called
User edits data	`CMyView` functions update `CMyDocument` data members
User exits application	`CMyView` object destroyed `CMyDocument` object destroyed

Table 1.

The `CFormView` Class

The `CFormView` class is a useful view class that has many of the characteristics of a modeless dialog window. Like a class derived from `CDialog`, a derived `CFormView` class is associated with a dialog resource that defines the frame characteristics and enumerates the controls. The `CFormView` class supports the same dialog data exchange and validation (DDX and DDV) functions that you saw in the `CDialog` examples in Module 5.
If AppWizard generates a **Form View** dialog, the properties are set correctly, but if you use the dialog editor to make a dialog for a form view, you must specify the following items in the **Dialog Properties** dialog:

1. **Style** = **Child**.
2. **Border** = **None**.
3. **Visible** = **unchecked**.

A `CFormView` object receives notification messages directly from its controls, and it receives command messages from the application framework. This application framework command-processing ability clearly separates `CFormView` from `CDialog` and it makes controlling the view from the frame's main menu or toolbar easy.
The `CFormView` class is derived from `CView` (actually, from `CScrollView`) and not from `CDialog`. You can't, therefore, assume that `CDialog` member functions are supported. `CFormView` does not have virtual `OnInitDialog()`, `OnOK()`, and `OnCancel()` functions. `CFormView()` member functions do not call `UpdateData()` and the DDX functions. You have to call `UpdateData()` yourself at the appropriate times, usually in response to control notification messages or command messages.
Even though the `CFormView` class is not derived from the `CDialog` class, it is built around the Microsoft Windows dialog. For this reason, you can use many of the `CDialog` class member functions such as `GotoDlgCtrl()` and `NextDlgCtrl()`. All you have to do is cast your `CFormView` pointer to a `CDialog` pointer. The following statement, extracted from a member function of a class derived from `CFormView`, sets the focus to a specified control. `GetDlgItem()` is a `CWnd` function and is thus inherited by the derived `CFormView` class.

```
((CDialog*) this)->GotoDlgCtrl(GetDlgItem(IDC_NAME));
```

AppWizard gives you the option of using `CFormView` as the base class for your view. When you select `CFormView`, AppWizard generates an empty dialog with the correct style properties set. The next step is to use ClassWizard to add control notification message handlers, command message handlers, and update command UI handlers. (The example steps starting after Figure 16-2 show you what to do.) You can also define data members and validation criteria.

The `CObject` Class

If you study the MFC library hierarchy, you'll notice that the `CObject` class is at the top. Most other classes are derived from the `CObject` root class. When a class is derived from `CObject`, it inherits a number of important characteristics. The many benefits of `CObject` derivation will become clear as you read the modules that follow. In this module, you'll see how `CObject` derivation allows objects to participate in the diagnostic dumping scheme and allows objects to be elements in the collection classes.

Diagnostic Dumping

The MFC library gives you some useful tools for diagnostic dumping. You enable these tools when you select the **Debug target**. When you select the **Win32 Release target**, diagnostic dumping is disabled and the diagnostic code is not linked to your program. All diagnostic output goes to the Debug view in the debugger's Output window. To clear diagnostic output from the debugger's Output window, position the cursor in the Output window and click the right mouse button. Then choose Clear from the pop-up menu.

The `TRACE` Macro

You've seen the `TRACE` macro used throughout the preceding examples in this book. `TRACE` statements are active whenever the constant `_DEBUG` is defined (when you select the Debug target and when the `afxTraceEnabled` variable is set to `TRUE`). `TRACE` statements work like C language **`printf`** statements, but they're completely disabled in the release version of the program. Here's a typical `TRACE` statement:

```
int nCount = 9;
CString strDesc("total");
TRACE("Count = %d, Description = %s\n", nCount, strDesc);
```

The `TRACE` macro takes a variable number of parameters and is thus easy to use. If you look at the MFC source code, you won't see `TRACE` macros but rather `TRACE0`, `TRACE1`, `TRACE2`, and `TRACE3` macros. These macros take 0, 1, 2, and 3 parameters, respectively, and are leftovers from the 16-bit environment, where it was necessary to conserve space in the data segment.

The `afxDump` Object

An alternative to the `TRACE` statement is more compatible with the C++ language. The MFC **`afxDump`** object accepts program variables with a syntax similar to that of **`cout`**, the C++ output stream object. You don't need complex formatting strings; instead, overloaded operators control the output format. The `afxDump` output goes to the same destination as the `TRACE` output, but the `afxDump` object is defined only in the Debug version of the MFC library. Here is a typical stream-oriented diagnostic statement that produces the same output as the `TRACE` statement above:

```
int nCount = 9;
CString strDesc("total");
#ifdef _DEBUG
    afxDump << "Count = " << nCount << ", Description = " << strDesc << "\n";
#endif // _DEBUG
```

Although both `afxDump` and cout use the same insertion operator (<<), they don't share any code. The cout object is part of the Microsoft Visual C++ **`iostream`** library, and `afxDump` is part of the MFC library. Don't assume that any of the cout formatting capability is available through `afxDump`.
Classes that aren't derived from `CObject`, such as `CString`, `CTime`, and `CRect`, contain their own overloaded insertion operators for `CDumpContext` objects. The `CDumpContext` class, of which `afxDump` is an instance,

includes the overloaded insertion operators for the native C++ data types (`int`, `double`, `char*`, and so on). The `CDumpContext` class also contains insertion operators for `CObject` references and pointers, and that's where things get interesting.

The Dump Context and the CObject Class

If the `CDumpContext` insertion operator accepts `CObject` pointers and references, it must also accept pointers and references to derived classes. Consider a trivial class, `CAction`, that is derived from `CObject`, as shown here:

```
class CAction : public CObject
{
  public:
    int m_nTime;
};
```

What happens when the following statement executes?

```
#ifdef _DEBUG
    afxDump << action; // action is an object of class CAction
#endif // _DEBUG
```

The virtual `CObject::Dump` function gets called. If you haven't overridden `Dump()` for `CAction`, you don't get much except for the address of the object. If you have overridden `Dump`, however, you can get the internal state of your object. Here's a `CAction::Dump` function:

```
#ifdef _DEBUG
void CAction::Dump(CDumpContext& dc) const
{
    CObject::Dump(dc); // Always call base class function
    dc << "time = " << m_nTime << "\n";
}
#endif // _DEBUG
```

The base class (`CObject`) `Dump()` function prints a line such as this:

```
a CObject at $4115D4
```

If you have called the `DECLARE_DYNAMIC` macro in your `CAction` class definition and the `IMPLEMENT_DYNAMIC` macro in your `CAction` declaration, you will see the name of the class in your dump:

```
a CAction at $4115D4
```

Even if your dump statement looks like this:

```
#ifdef _DEBUG
    afxDump << (CObject&) action;
#endif // _DEBUG
```

The two macros work together to include the MFC library runtime class code in your derived `CObject` class. With this code in place, your program can determine an object's class name at runtime (for the dump, for example) and it can obtain class hierarchy information. The (`DECLARE_SERIAL`, `IMPLEMENT_SERIAL`) and (`DECLARE_DYNCREATE`, `IMPLEMENT_DYNCREATE`) macro pairs provide the same runtime class features as those provided by the (`DECLARE_DYNAMIC`, `IMPLEMENT_DYNAMIC`) macro pair.

Automatic Dump of Undeleted Objects

With the **Debug target** selected, the application framework dumps all objects that are undeleted when your program exits. This dump is a useful diagnostic aid, but if you want it to be really useful, you must be sure to delete all your objects, even the ones that would normally disappear after the exit. This object cleanup is good programming discipline. The code that adds debug information to allocated memory blocks is now in the Debug version of the CRT (C runtime)

library rather than in the MFC library. If you choose to dynamically link MFC, the MSVCRTD DLL is loaded along with the necessary MFC DLLs. When you add the line:

```
#define new DEBUG_NEW
```

at the top of a CPP file, the CRT library lists the filename and line number at which the allocations were made. AppWizard puts this line at the top of all the CPP files it generates.

Window Subclassing for Enhanced Data-Entry Control

What if you want an edit control (in a dialog or a form view) that accepts only numeric characters? That's easy. You just set the Number style in the control's property sheet. If, however, you want to exclude numeric characters or change the case of alphabetic characters, you must do some programming.
The MFC library provides a convenient way to change the behavior of any standard control, including the edit control. Actually, there are several ways. You can derive your own classes from CEdit, CListBox, and so forth (with their own message handler functions) and then create control objects at runtime. Or you can register a special window class, as a Win32 programmer would do, and integrate it into the project's resource file with a text editor. Neither of these methods, however, allows you to use the dialog editor to position controls in the dialog resource.
The easy way to modify a control's behavior is to use the MFC library's window subclassing feature. You use the dialog editor to position a normal control in a dialog resource, and then you write a new C++ class that contains message handlers for the events that you want to handle yourself.

Here are the steps for subclassing an edit control:

With the dialog editor, position an edit control in your dialog resource. Assume that it has the child window ID IDC_EDIT1. Write a new class, for example, CNonNumericEdit, derived from CEdit. Map the WM_CHAR message and write a handler like this:

```
void CNonNumericEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if (!isdigit(nChar))
    {
        CEdit::OnChar(nChar, nRepCnt, nFlags);
    }
}
```

In your derived dialog or form view class header, declare a data member of class CNonNumericEdit in this way:

```
private:
    CNonNumericEdit m_nonNumericEdit;
```

If you're working with a dialog class, add the following line to your OnInitDialog() override function:

```
m_nonNumericEdit.SubclassDlgItem(IDC_EDIT1, this);
```

If you're working with a form view class, add the following code to your OnInitialUpdate() override function:

```
if (m_nonNumericEdit.m_hWnd == NULL)
{
    m_nonNumericEdit.SubclassDlgItem(IDC_EDIT1, this);
}
```

The CWnd::SubclassDlgItem member function ensures that all messages are routed through the application framework's message dispatch system before being sent to the control's built-in window procedure. This technique is called dynamic subclassing and is explained in more detail in Technical Note #14 in the online documentation. The code in the preceding steps only accepts or rejects a character. If you want to change the value of a character, your handler must call CWnd::DefWindowProc, which bypasses some MFC logic that stores parameter values in thread object data members. Here's a sample handler that converts lowercase characters to uppercase:

```
void CUpperEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
```

```
{
    if (islower(nChar))
    {
        nChar = toupper(nChar);
    }
    DefWindowProc(WM_CHAR, (WPARAM) nChar, (LPARAM) (nRepCnt | (nFlags << 16)));
}
```

You can also use window subclassing to handle reflected messages, which were mentioned in Module 5. If an MFC window class doesn't map a message from one of its child controls, the framework reflects the message back to the control. Technical Note #62 in the online documentation explains the details.
If you need an edit control with a yellow background, for example, you can derive a class `CYellowEdit` from `CEdit` and use ClassWizard to map the `WM_CTLCOLOR` message in `CYellowEdit`. ClassWizard lists the message name with an equal sign in front to indicate that it is reflected. The handler code, shown below, is substantially the same as the non-reflected `WM_CTLCOLOR` handler. Member variable `m_hYellowBrush` is defined in the control class's constructor.

```
HBRUSH CYellowEdit::CtlColor(CDC* pDC, UINT nCtlColor)
{
    pDC->SetBkColor(RGB(255, 255, 0)); // yellow
    return m_hYellowBrush;
}
```

The MYMFC15 Example

The first of this module's two examples shows a very simple **document-view interaction**. The `CMymfc15Doc` document class, derived from `CDocument`, allows for a single embedded `CStudent` object. The `CStudent` class represents a student record composed of a `CString` name and an integer grade. The `CMymfc15View` view class is derived from `CFormView`. It is a visual representation of a student record that has edit controls for the name and grade. The default Enter pushbutton updates the document with data from the edit controls. Figure 1 shows the MYMFC15 program window.

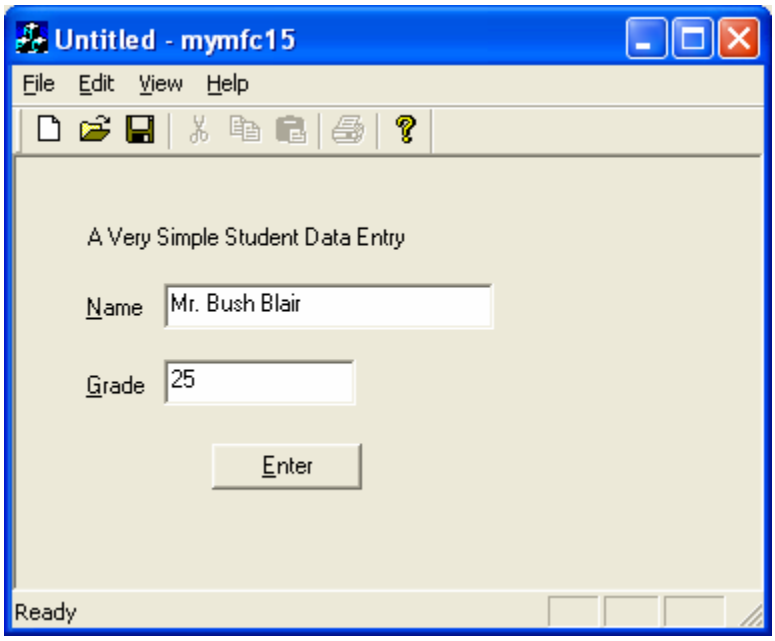


Figure 1: The MYMFC15 program in action.

Listing 1 shows the code for the `CStudent` class (**Student.h** and **Student.cpp**). Most of the class's features serve MYMFC15, but a few items carry forward to MYMFC16 and the programs discussed in Module 11. For now, take note of the two data members, the default constructor, the operators, and the `Dump()` function declaration. The `DECLARE_DYNAMIC` and `IMPLEMENT_DYNAMIC` macros ensure that the class name is available for the diagnostic dump.

STUDENT.H

```
// student.h

#ifndef _INSIDE_VISUAL_CPP_STUDENT
#define _INSIDE_VISUAL_CPP_STUDENT
class CStudent : public CObject
{
    DECLARE_DYNAMIC(CStudent)
public:
    CString m_strName;
    int m_nGrade;

    CStudent()
    {
        m_nGrade = 0;
    }

    CStudent(const char* szName, int nGrade) : m_strName(szName)
    {
        m_nGrade = nGrade;
    }

    CStudent(const CStudent& s) : m_strName(s.m_strName)
    {
        // copy constructor
        m_nGrade = s.m_nGrade;
    }

    const CStudent& operator =(const CStudent& s)
    {
        m_strName = s.m_strName;
        m_nGrade = s.m_nGrade;
        return *this;
    }

    BOOL operator ==(const CStudent& s) const
    {
        if ((m_strName == s.m_strName) && (m_nGrade == s.m_nGrade)) {
            return TRUE;
        }
        else {
            return FALSE;
        }
    }

    BOOL operator !=(const CStudent& s) const
    {
        // Let's make use of the operator we just defined!
        return !(*this == s);
    }
#ifdef _DEBUG
    void Dump(CDumpContext& dc) const;
#endif // _DEBUG
};

#endif // _INSIDE_VISUAL_CPP_STUDENT
```

STUDENT.CPP

```
#include "stdafx.h"
#include "student.h"

IMPLEMENT_DYNAMIC(CStudent, CObject)

#ifdef _DEBUG
```

```
void CStudent::Dump(CDumpContext& dc) const
{
    CObject::Dump(dc);
    dc << "m_strName = " << m_strName << "\nm_nGrade = " <<m_nGrade;
}
#endif // _DEBUG
```

Listing 1: The `CStudent` class listing.

Follow these steps to build the MYMFC15 example:

Run AppWizard to generate \mfcproject\mymfc15. In the Step 6 page, change the view's base class to `CFormView`, as shown here.

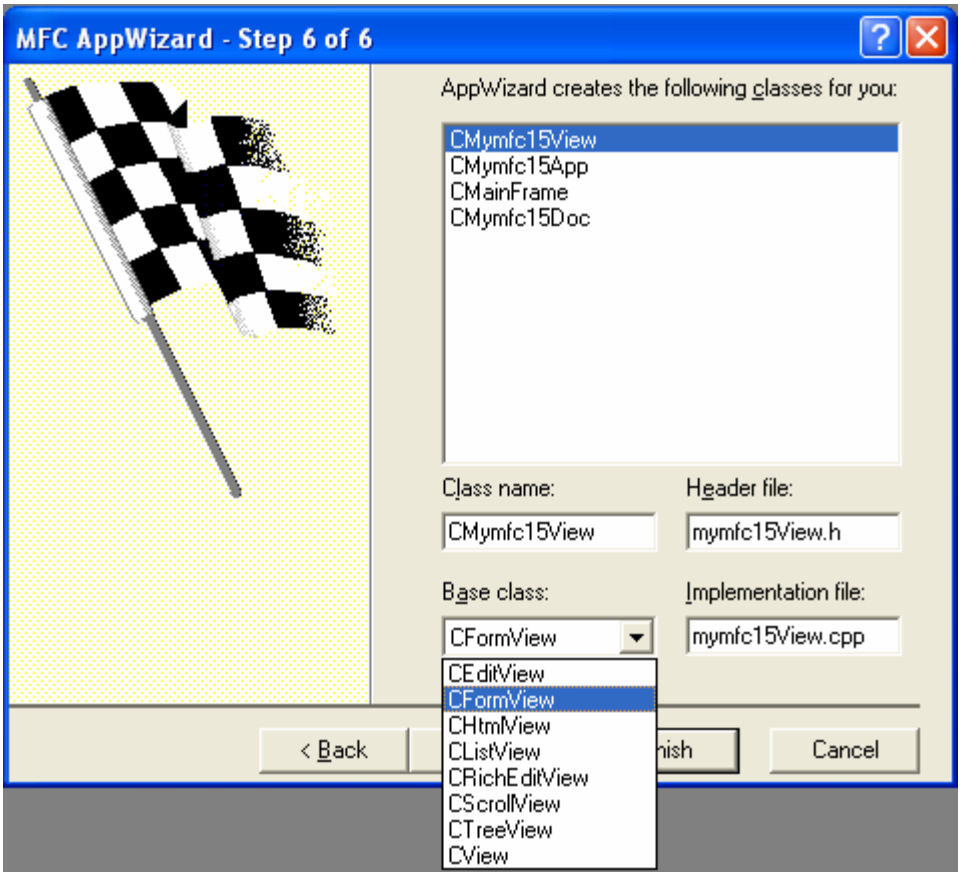


Figure 2: Step 6 of 6 AppWizard, changing the view base class to `CFormView` class.

The options and the default class names are shown here.

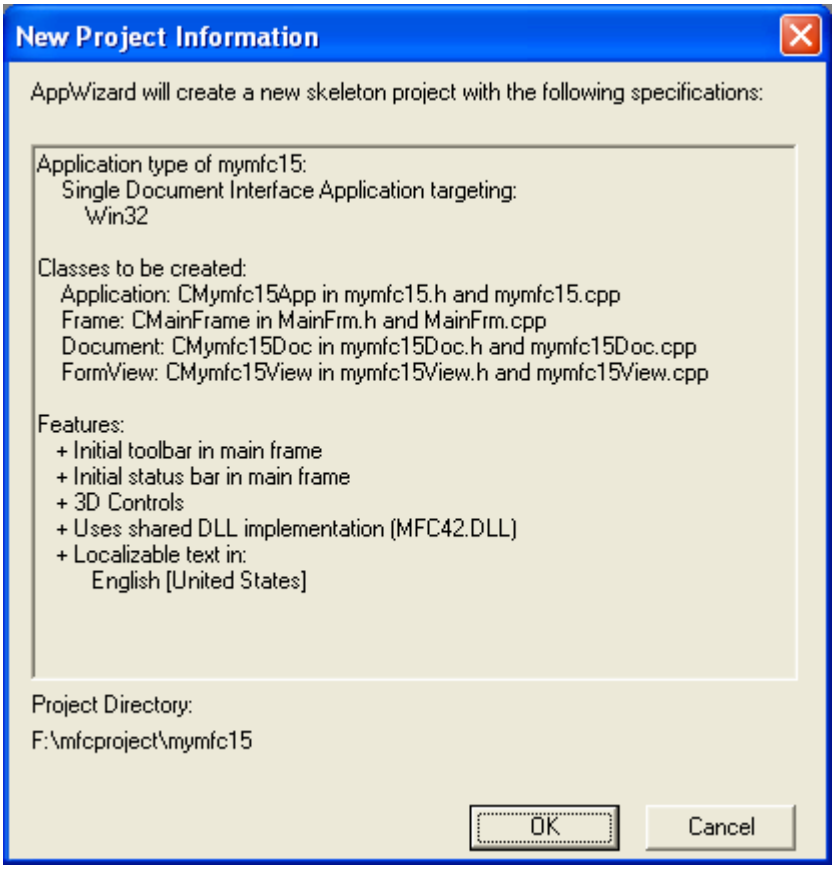


Figure 3: MYMFC15 project summary.

Use the menu editor to replace the **Edit** menu options. Delete the current **Edit** menu items and replace them with a **Clear All** option, as shown here.

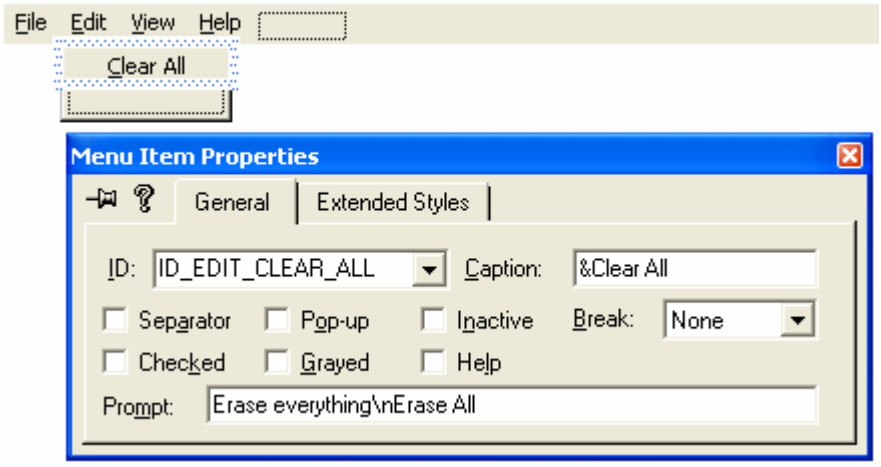


Figure 4: Adding and modifying the **Edit** menu properties.

Use the default constant `ID_EDIT_CLEAR_ALL`, which is assigned by the application framework. A menu prompt automatically appears.

Use the dialog editor to modify the `IDD_MYMFC15_FORM` dialog. Open the AppWizard-generated dialog `IDD_MYMFC15_FORM`, and add controls as shown below. Be sure that the **Styles** properties are set exactly as shown in the **Dialog Properties** dialog (**Style** = **Child**; **Border** = **None**) and that **Visible** is unchecked.

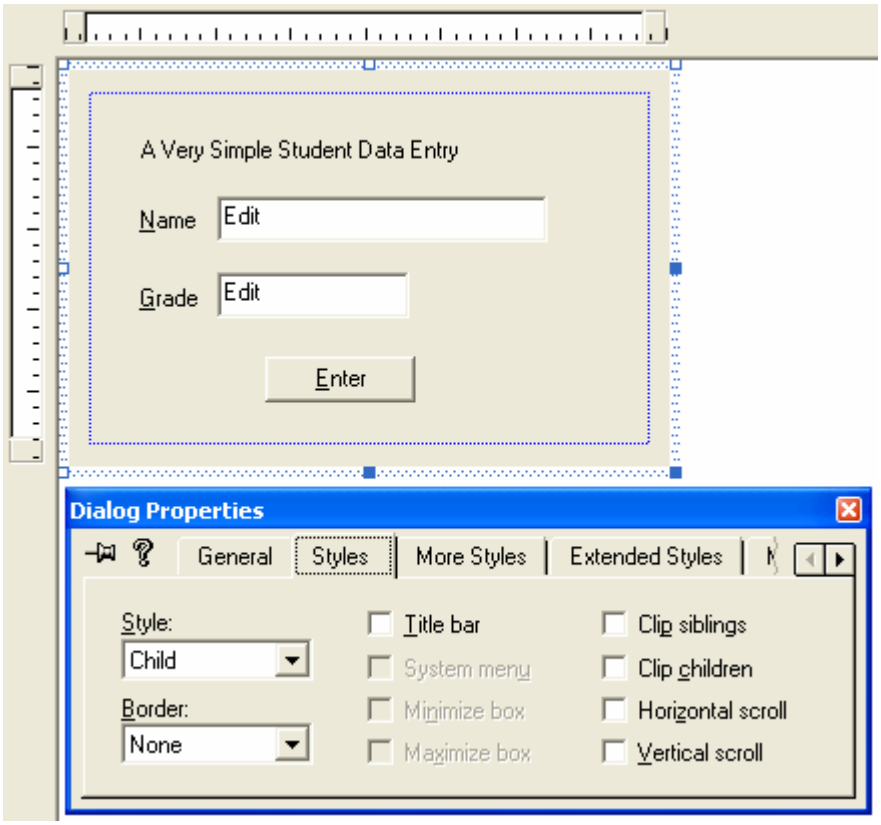


Figure 5: Modifying the `IDD_MYMFC15_FORM` dialog and its properties.

Use the following IDs for the controls.

Control	ID
Name edit control	`IDC_NAME`
Grade edit control	`IDC_GRADE`
Enter pushbutton	`IDC_ENTER`

Table 2.

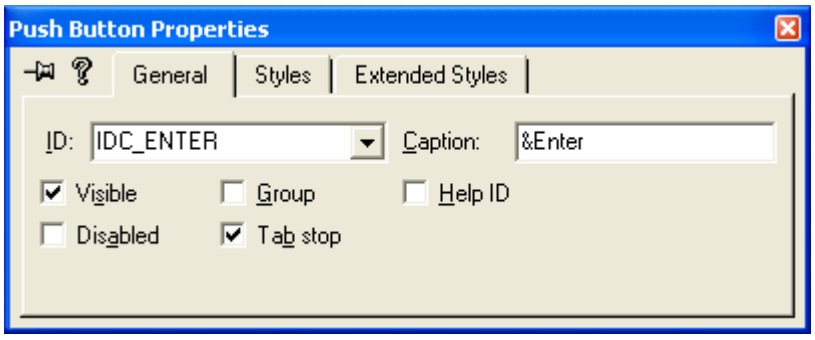


Figure 6: Modifying the push button properties.

Use ClassWizard to add message handlers for `CMymfc15View`. Select the `CMymfc15View` class, and then add handlers for the following messages. Accept the default function names.

Object ID	Message	Member Function
`IDC_ENTER`	`BN_CLICKED`	`OnEnter()`

ID_EDIT_CLEAR_ALL	COMMAND	OnEditClearAll()
ID_EDIT_CLEAR_ALL	UPDATE_COMMAND_UI	OnUpdateEditClearAll()

Table 3.

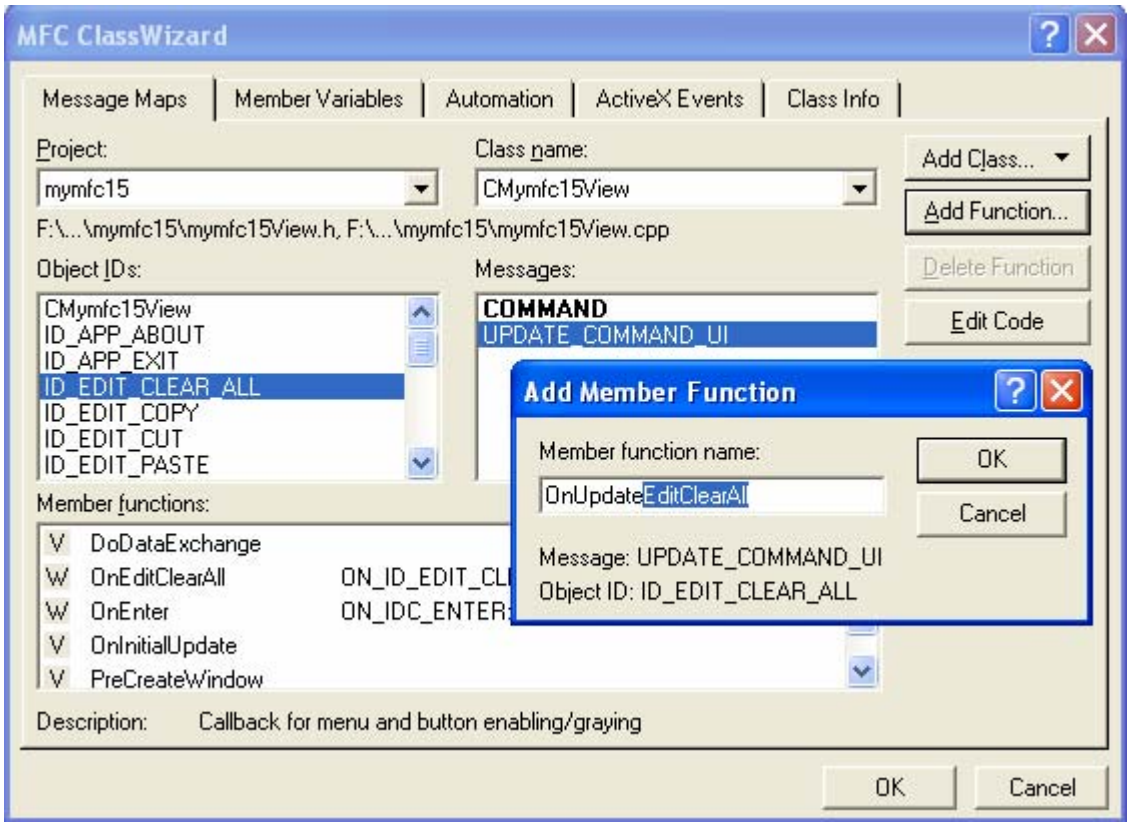


Figure 7: Using ClassWizard to add message handlers for `CMymfc15View`.

Use ClassWizard to add variables for `CMymfc15View`. Click on the **Member Variables** tab in the MFC ClassWizard dialog, and then add the following variables.

Control ID	Member Variable	Category	Variable Type
IDC_GRADE	m_nGrade	Value	int
IDC_NAME	m_strName	Value	CString

Table 4.

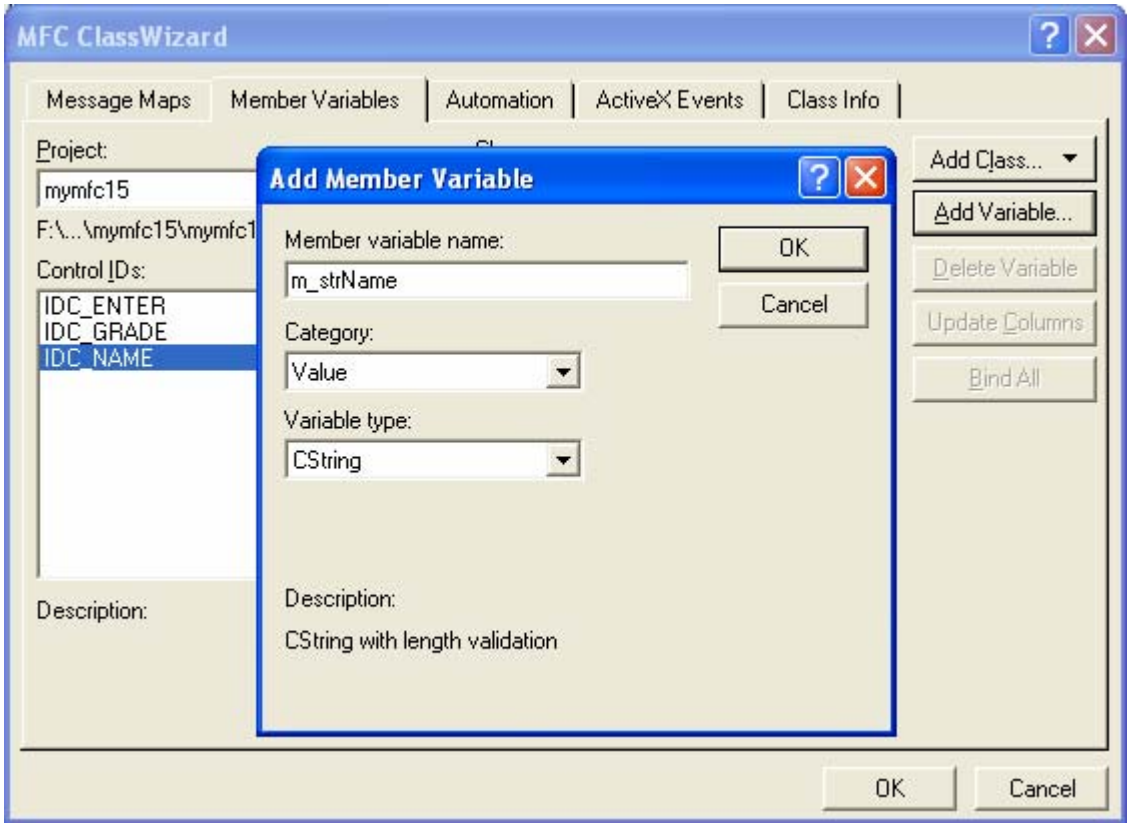


Figure 8: Using ClassWizard to add variables for `CMymfc15View`..

For `m_nGrade`, enter a minimum value of **0** and a maximum value of **100**.

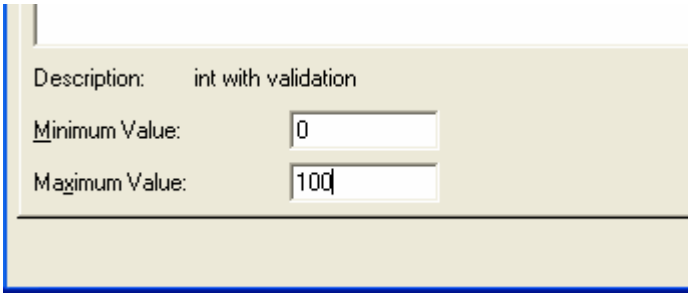


Figure 9: Setting the minimum and maximum value for `m_nGrade`.

Notice that ClassWizard generates the code necessary to validate data entered by the user.

```
void CMymfc15View::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CMymfc15View)
    DDX_Text(pDX, IDC_GRADE, m_nGrade);
    DDV_MinMaxInt(pDX, m_nGrade, 0, 100);
    DDX_Text(pDX, IDC_NAME, m_strName);
    //}}AFX_DATA_MAP
}
```

Listing 2.

Add a prototype for the helper function `UpdateControlsFromDoc()`. In the ClassView window, right-click on `CMymfc15View` and choose **Add Member Function**. Fill out the dialog box to add the following function:

```
private:
    void UpdateControlsFromDoc();
```

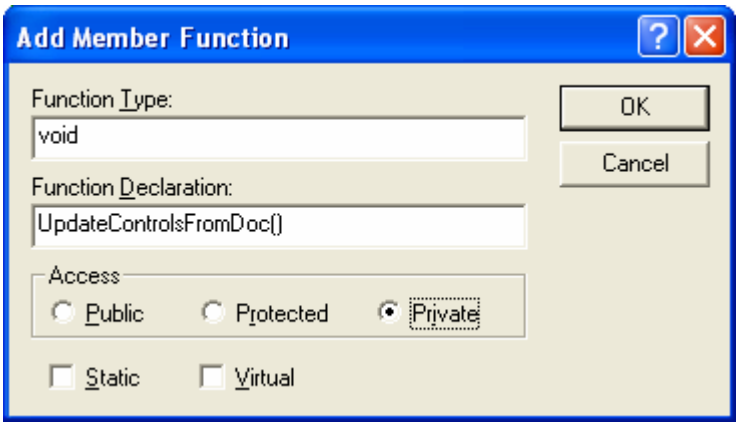


Figure 10: Using ClassView to add a prototype for the helper function `UpdateControlsFromDoc()`.

Edit the file **Mymfc15View.cpp**. AppWizard generated the skeleton `OnInitialUpdate()` function and ClassView generated the skeleton `UpdateControlsFromDoc()` function. `UpdateControlsFromDoc()` is a private helper member function that transfers data from the document to the `CMymfc15View` data members and then to the dialog edit controls. Edit the code as shown here:

```
void CMymfc15View::OnInitialUpdate()
{   // called on startup
    UpdateControlsFromDoc();
}
```

```
void CMymfc15View::OnInitialUpdate()
{
    // called on startup
    UpdateControlsFromDoc();
}

/////////////////////////////////////
// CMymfc15View diagnostics
```

Listing 3.

```
void CMymfc15View::UpdateControlsFromDoc()
{   // called from OnInitialUpdate and OnEditClearAll
    CMymfc15Doc* pDoc = GetDocument();
    m_nGrade = pDoc->m_student.m_nGrade;
    m_strName = pDoc->m_student.m_strName;

    UpdateData(FALSE); // calls DDX
}
```

```
void CMymfc15View::UpdateControlsFromDoc()
{
    // called from OnInitialUpdate and OnEditClearAll
    CMymfc15Doc* pDoc = GetDocument();
    m_nGrade = pDoc->m_student.m_nGrade;
    m_strName = pDoc->m_student.m_strName;

    UpdateData(FALSE); // calls DDX
}
```

Listing 4.

The `OnEnter()` function replaces the `OnOK()` function you'd expect to see in a dialog class. The function transfers data from the edit controls to the view's data members and then to the document. Add the code shown here:

```
void CMymfc15View::OnEnter()
{
    CMymfc15Doc* pDoc = GetDocument();
    UpdateData(TRUE);
    pDoc->m_student.m_nGrade = m_nGrade;
    pDoc->m_student.m_strName = m_strName;
}
```

```
void CMymfc15View::OnEnter()
{
    // TODO: Add your control notification handler code here
    CMymfc15Doc* pDoc = GetDocument();
    UpdateData(TRUE);
    pDoc->m_student.m_nGrade = m_nGrade;
    pDoc->m_student.m_strName = m_strName;
}
```

Listing 5.

In a complex multi-view application, the **Edit Clear All** command would be routed directly to the document. In this simple example, it's routed to the view. The update command UI handler disables the menu item if the document's student object is already blank. Add the following code:

```
void CMymfc15View::OnEditClearAll()
{
    // "blank" student object
    GetDocument()->m_student = CStudent();
    UpdateControlsFromDoc();
}

void CMymfc15View::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
      // blank?
      pCmdUI->Enable(GetDocument()->m_student != CStudent());
}
```

```
void CMymfc15View::OnEditClearAll()
{
    // TODO: Add your command handler code here
    // "blank" student object
    GetDocument()->m_student = CStudent();
    UpdateControlsFromDoc();
}

void CMymfc15View::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    // blank?
    pCmdUI->Enable(GetDocument()->m_student != CStudent());
}
```

Listing 6.

Edit the MYMFC15 project to add the files for `CStudent`. Choose **Add To Project** from the **Project** menu, choose **New** from the submenu, and select the **C/C++ Header File** and type the **Student** as the file name.

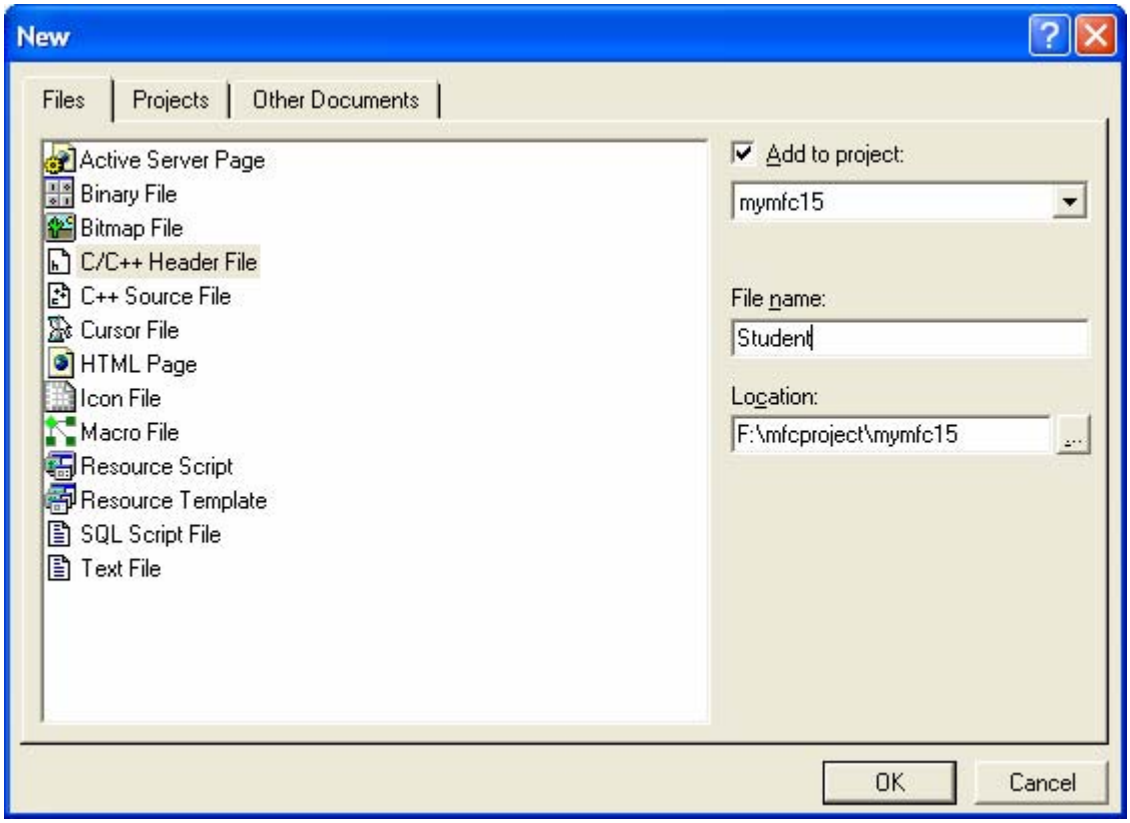


Figure 11: Adding new header file to the project for the `Student` class.

Then copy the previous **Student.h** code into the newly created **Student.h** file. Repeat the similar steps for **Student.cpp** file. Select the **C++ Source File** in this case.

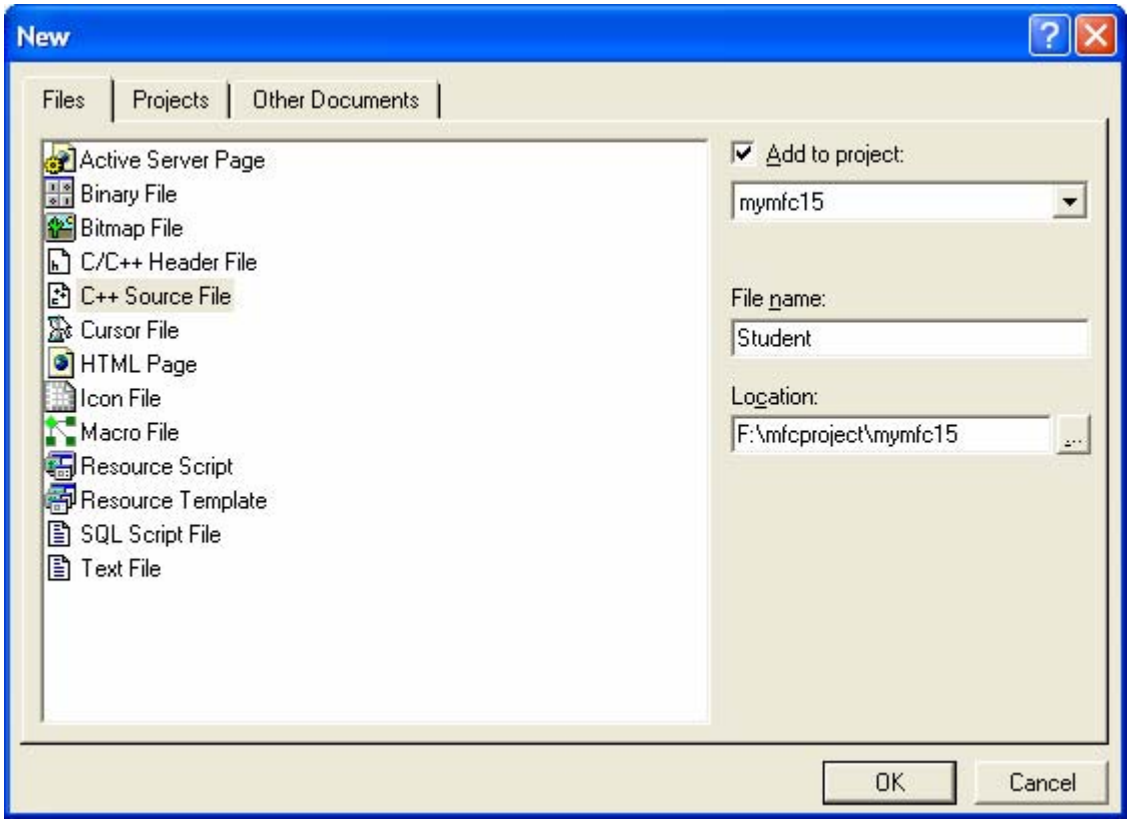


Figure 12: Adding new source file to the project for the `Student` class.

Visual C++ will add the files' names to the project's DSP file so that they will be compiled when you build the project. Add a `CStudent` data member to the `CMymfc15Doc` class. Use ClassView to add the following data member, and the `#include` will be added automatically.

```
public:
    CStudent m_student;
```

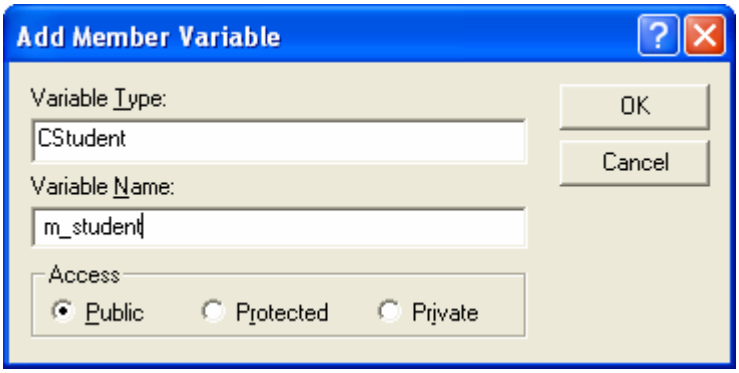


Figure 13: Adding a `CStudent` data member to the `CMymfc15Doc` class.

The `CStudent` constructor is called when the document object is constructed, and the `CStudent` destructor is called when the document object is destroyed. Edit the **Mymfc15Doc.cpp** file. Use the `CMymfc15Doc` constructor to initialize the student object, as shown here:

```
CMymfc15Doc::CMymfc15Doc() : m_student("The default value", 0)
{
    TRACE("Document object constructed\n");
```

```
}

CMymfc15Doc::CMymfc15Doc() : m_student("The default value", 0)
{
    // TODO: add one-time construction code here
    TRACE("Document object constructed\n");
}
```

Listing 7.

We can't tell whether the MYMFC15 program works properly unless we dump the document when the program exits. We'll use the destructor to call the document's `Dump()` function, which calls the `CStudent::Dump` function shown here:

```
CMymfc15Doc::~CMymfc15Doc()
{
      #ifdef _DEBUG
          Dump(afxDump);
      #endif // _DEBUG
}

CMymfc15Doc::~CMymfc15Doc()
{
    #ifdef _DEBUG
    Dump(afxDump);
    #endif // _DEBUG
}
```

Listing 8.

```
void CMymfc15Doc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
    dc << "\n" << m_student << "\n";
}


// CMymfc15Doc diagnostics

#ifdef _DEBUG
void CMymfc15Doc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMymfc15Doc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
    dc << "\n" << m_student << "\n";
}
#endif //_DEBUG

////////////////////////////////////////////////////
// CMymfc15Doc commands
```

Listing 9.

Build and test the MYMFC15 application. Type a name and a grade, and then click **Enter**. Now exit the application.

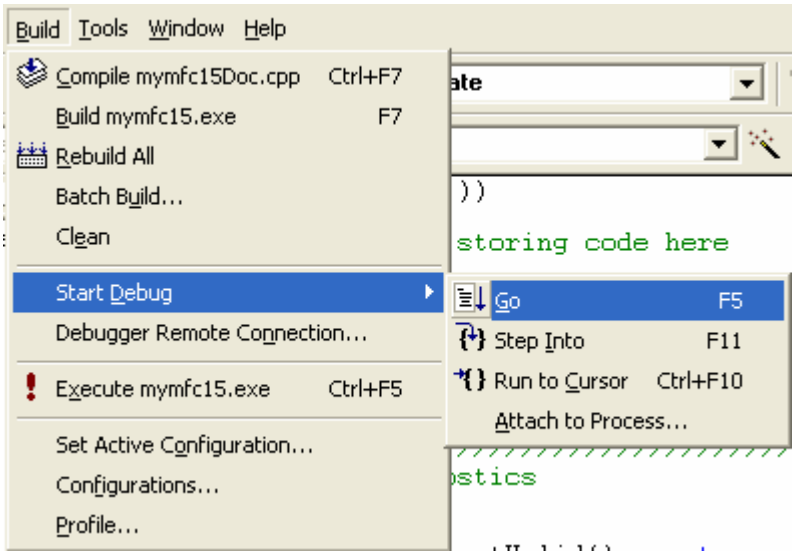


Figure 14: Building MYMFC15 in debug mode.

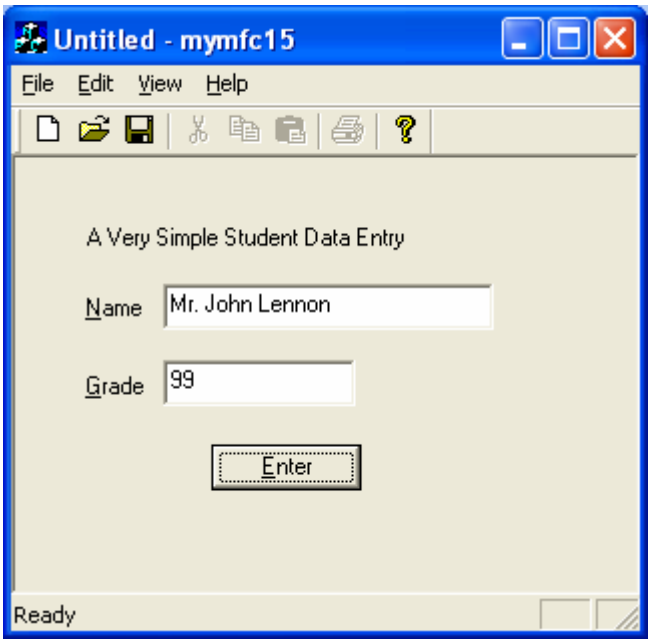
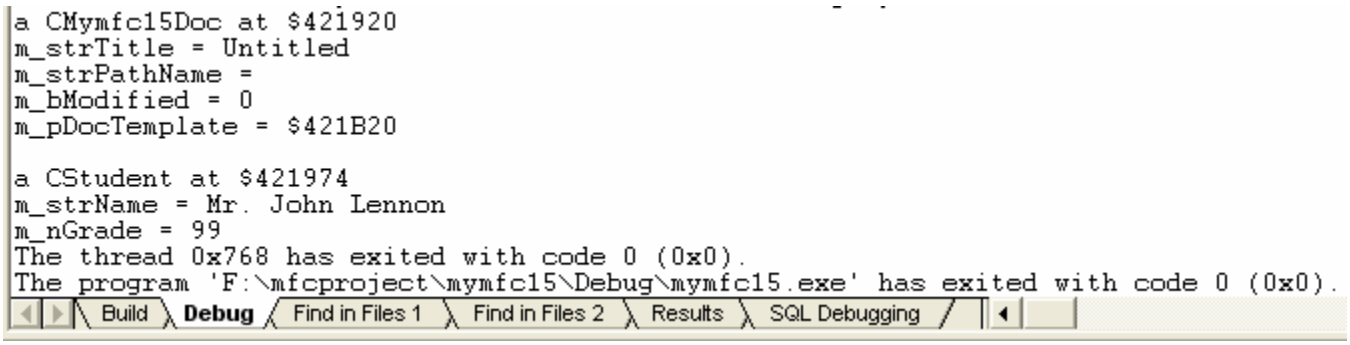


Figure 15: MYMFC15 program output, testing the functionalities.

Does the Debug window show messages similar to those shown here?

```
...
a CMymfc15Doc at $421920
m_strTitle = Untitled
m_strPathName =
m_bModified = 0
m_pDocTemplate = $421B20

a CStudent at $421974
m_strName = Mr. John Lennon
m_nGrade = 99
...
```



Listing 10.

To see these messages, you must compile the application with the Win32 Debug target selected and you must run the program from the debugger.

A More Advanced Document-View Interaction

If you're laying the groundwork for a multiview application, the document-view interaction must be more complex than the simple interaction in example MYMFC15. The fundamental problem is this: the user edits in view #1, so view #2 (and any other views) must be updated to reflect the changes. Now you need the `UpdateAllViews()` and `OnUpdate()` functions because the document is going to act as the clearinghouse for all view updates. The development steps are shown here:

1. In your derived document class header file (generated by AppWizard), declare your document's data members. If you want to, you can make these data members private and you can define member functions to access them or declare the view class as a friend of the document class.
2. In your derived view class, use ClassWizard to override the `OnUpdate()` virtual member function. The application framework calls this function whenever the document data has changed for any reason. `OnUpdate()` should update the view with the current document data.
3. Evaluate all your command messages. Determine whether each one is document-specific or view-specific. A good example of a document-specific command is the **Clear All** command on the **Edit** menu. Now map the commands to the appropriate classes.
4. In your derived view class, allow the appropriate command message handlers to update the document data. Be sure these message handlers call the `CDocument::UpdateAllViews` function before they exit. Use the type-safe version of the `CView::GetDocument` member function to access the view's document.
5. In your derived document class, allow the appropriate command message handlers to update the document data. Be sure that these message handlers call the `CDocument::UpdateAllViews` function before they exit.

The sequence of events for the complex document-view interaction is shown here.

Sequence	Description
Application starts	`CMyDocument` object constructed `CMyView` object constructed Other view objects constructed `View` windows created `CMyView::OnCreate` called (if mapped) `CDocument::OnNewDocument` called `CView::OnInitialUpdate` called Calls `CMyView::OnUpdate` Initializes the view
User executes view command	`CMyView` functions update `CMyDocument` data members Call `CDocument::UpdateAllViews` Other views' `OnUpdate()` functions called
User executes document command	`CMyDocument` functions update data members Call `CDocument::UpdateAllViews`

	`CMyView::OnUpdate` called Other views' `OnUpdate()` functions called
User exits application	View objects destroyed `CMyDocument` object destroyed

Table 5.

The `CDocument::DeleteContents` Function

At some point, you'll need a function to delete the contents of your document. You could write your own private member function, but it happens that the application framework declares a virtual `DeleteContents()` function for the `CDocument` class. The application framework calls your overridden `DeleteContents()` function when the document is closed and as you'll see in the next module, at other times as well.

The `CObList` Collection Class

Once you get to know the collection classes, you'll wonder how you ever got along without them. The `CObList` class is a useful **representative of the collection class family**. If you're familiar with this class, it's easy to learn the other list classes, the array classes, and the map classes.
You might think that collections are something new, but the C programming language has always supported one kind of collection, the **array**. C arrays must be fixed in size, and they do not support insertion of elements. Many C programmers have written function libraries for other collections, including linked lists, dynamic arrays, and indexed dictionaries. For implementing collections, the C++ class is an obvious and better alternative than a C function library. A list object, for example, neatly encapsulates the list's internal data structures.
The `CObList` class supports ordered lists of pointers to objects of classes derived from `CObject`. Another MFC collection class, `CPtrList`, stores void pointers instead of `CObject` pointers. Why not use `CPtrList` instead? The `CObList` class offers advantages for diagnostic dumping, which you'll see in this chapter, and for serialization, which you'll see in the next chapter. One important feature of `CObList` is that it can contain mixed pointers. In other words, a `CObList` collection can hold pointers to both `CStudent` objects and `CTeacher` objects, assuming that both `CStudent` and `CTeacher` were derived from `CObject`.

Using the `CObList` Class for a First-In, First-Out List

One of the easiest ways to use a `CObList` object is to add new elements to the tail, or bottom, of the list and to remove elements from the head, or top, of the list. The first element added to the list will always be the first element removed from the head of the list. Suppose you're working with element objects of class `CAction`, which is your own custom class derived from `CObject`. A command-line program that puts five elements into a list and then retrieves them in the same sequence is shown here:

```
#include <afx.h>
#include <afxcoll.h>

class CAction : public CObject
{
private:
    int m_nTime;
public:
    // Constructor stores integer time value
    CAction(int nTime) { m_nTime = nTime; }
    void PrintTime() { TRACE("time = %d\n", m_nTime); }
};

int main()
{
    CAction* pAction;
    // action list constructed on stack
    CObList actionList;
    int i;
```

```
    // inserts action objects in sequence {0, 1, 2, 3, 4}
    for (i = 0; i < 5; i++)
    {
        pAction = new CAction(i);
        // no cast necessary for pAction
        actionList.AddTail(pAction);
    }

    // retrieves and removes action objects
    // in sequence {0, 1, 2, 3, 4}
    while (!actionList.IsEmpty())
    {
        // cast required for return value
        pAction = (CAction*) actionList.RemoveHead();
        pAction->PrintTime();
        delete pAction;
    }

    return 0;
}
```

Here's what's going on in the program. First a `CObList` object, `actionList`, is constructed. Then the `CObList::AddTail` member function inserts pointers to newly constructed `CAction` objects. No casting is necessary for `pAction` because `AddTail()` takes a `CObject` pointer parameter and `pAction` is a pointer to a derived class. Next the `CAction` object pointers are removed from the list of the objects deleted. A cast is necessary for the returned value of `RemoveHead()` because `RemoveHead()` returns a `CObject` pointer that is higher in the class hierarchy than `CAction`. When you remove an object pointer from a collection, the object is not automatically deleted. The delete statement is necessary for deleting the `CAction` objects.

`CObList` Iteration: The `POSITION` Variable

Suppose you want to iterate through the elements in a list. The `CObList` class provides a `GetNext()` member function that returns a pointer to the "next" list element, but using it is a little tricky. `GetNext()` takes a parameter of type `POSITION`, which is a 32-bit variable. The `POSITION` variable is an internal representation of the retrieved element's position in the list. Because the `POSITION` parameter is declared as a reference (`&`), the function can change its value.

`GetNext()` does the following:

1. It returns a pointer to the "current" object in the list, identified by the incoming value of the `POSITION` parameter.
2. It increments the value of the `POSITION` parameter to the next list element.

Here's what a `GetNext()` loop looks like, assuming you're using the list generated in the previous example:

```
CAction* pAction;
POSITION pos = actionList.GetHeadPosition();
while (pos != NULL)
{
    pAction = (CAction*) actionList.GetNext(pos);
    pAction->PrintTime();
}
```

Now suppose you have an interactive Windows-based application that uses toolbar buttons to sequence forward and backward through the list one element at a time. You can't use `GetNext()` to retrieve the entry because `GetNext()` always increments the `POSITION` variable and you don't know in advance whether the user is going to want the next element or the previous element. Here's a sample view class command message handler function that gets the next list entry. In the `CMyView` class, `m_actionList` is an embedded `CObList` object and the `m_position` data member is a `POSITION` variable that holds the current list position.

```
CMyView::OnCommandNext()
{
    POSITION pos;
    CAction*  pAction;

    if ((pos = m_position) != NULL)
    {
        m_actionList.GetNext(pos);
        if (pos != NULL)
        { // pos is NULL at end of list
            pAction = (CAction*) m_actionList.GetAt(pos);
            pAction->PrintTime();
            m_position = pos;
        }
        else
        {
            AfxMessageBox("End of list reached");
        }
    }
}
```

`GetNext()` is now called first to increment the list position, and the `CObList::GetAt` member function is called to retrieve the entry. The `m_position` variable is updated only when we're sure we're not at the tail of the list.

The `CTypedPtrList` Template Collection Class

The `CObList` class works fine if you want a collection to contain mixed pointers. If, on the other hand, you want a type-safe collection that contains only one type of object pointer, you should look at the MFC library template pointer collection classes. `CTypedPtrList` is a good example. Templates are a relatively new C++ language element, introduced by Microsoft Visual C++ version 2.0. `CTypedPtrList` is a template class that you can use to create a list of any pointers to objects of any specified class. To make a long story short, you use the template to create a custom derived list class, using either `CPtrList` or `CObList` as a base class. To declare an object for `CAction` pointers, you write the following line of code:

```
CTypedPtrList<CObList, CAction*> m_actionList;
```

The first parameter is the base class for the collection, and the second parameter is the type for parameters and return values. Only `CPtrList` and `CObList` are permitted for the base class because those are the only two MFC library pointer list classes. If you are storing objects of classes derived from `CObject`, you should use `CObList` as your base class; otherwise, use `CPtrList`. By using the template as shown above, the compiler ensures that all list member functions return a `CAction` pointer. Thus, you can write the following code:

```
pAction = m_actionList.GetAt(pos); // no cast required
```

If you want to clean up the notation a little, use a `typedef` statement to generate what looks like a class, as shown here:

```
typedef CTypedPtrList<CObList, CAction*> CActionList;
```

Now you can declare `m_actionList` as follows:

```
CActionList m_actionList;
```

The Dump Context and Collection Classes

The `Dump()` function for `CObList` and the other collection classes has a useful property. If you call `Dump()` for a collection object, you can get a display of each object in the collection. If the element objects use the `DECLARE_DYNAMIC` and `IMPLEMENT_DYNAMIC` macros, the dump will show the class name for each object. The default behavior of the collection `Dump()` functions is to display only class names and addresses of element objects. If you want the collection `Dump()` functions to call the `Dump()` function for each element object, you must, somewhere at the start of your program, make the following call:

```
#ifdef _DEBUG
    afxDump.SetDepth(1);
#endif
```

Now the statement:

```
#ifdef _DEBUG
    afxDump << actionList;
#endif
```

Produces output such as this:

```
a CObList at $411832
with 4 elements
    a CAction at $412CD6
time = 0
    a CAction at $412632
time = 1
    a CAction at $41268E
time = 2
    a CAction at $4126EA
time = 3
```

If the collection contains mixed pointers, the virtual `Dump()` function is called for the object's class and the appropriate class name is printed.

The MYMFC16 Example

Run AppWizard to generate SDI \mfcproject\mymfc16 project. In the Step 6 page, change the view's base class to `CFormView`, as shown here.

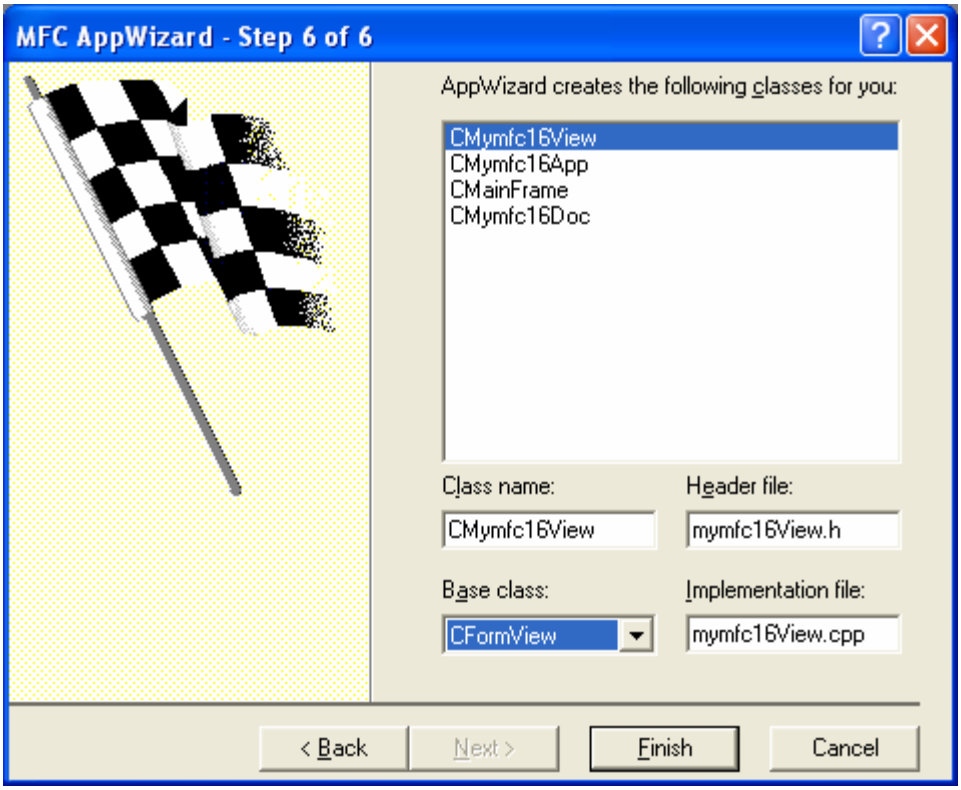


Figure 16: AppWizard step 6 of 6, changing view's base class to `CFormView`.

The options and the default class names are shown here.

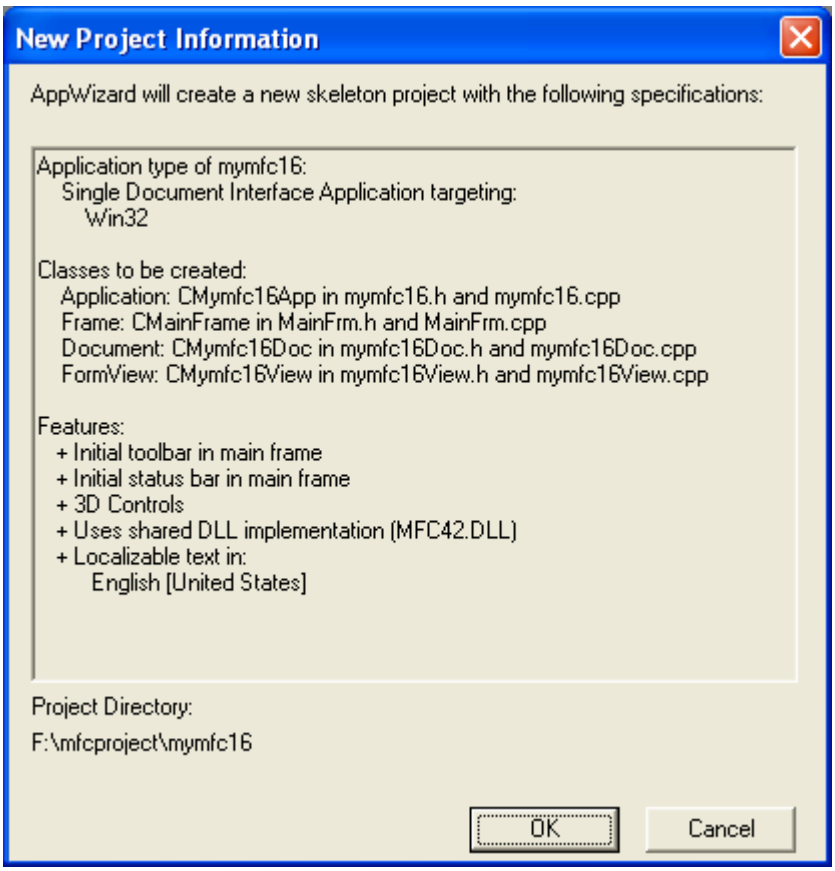


Figure 17: MYMFC16 project summary.

This SDI example improves on MYMFC15 in the following ways:

- Instead of a single embedded `CStudent` object, the document now contains a list of `CStudent` objects. Now you see the reason for using the `CStudent` class instead of making `m_strName` and `m_nGrade` data members of the document.
- Toolbar buttons allow the user to sequence through the list.
- The application is structured to allow the addition of extra views. The **Edit Clear All** command is now routed to the document object, so the document's `UpdateAllViews()` function and the view's `OnUpdate()` function are brought into play.
- The student-specific view code is isolated so that the `CMymfc16View` class can later be transformed into a base class that contains only general-purpose code. Derived classes can override selected functions to accommodate lists of application-specific objects.

The MYMFC16 window, shown in Figure 18, looks a little different from the MYMFC15 window shown in Figure 1. The toolbar buttons are enabled only when appropriate. The **Next** (arrow-down graphic) button, for example, is disabled when we're positioned at the bottom of the list.

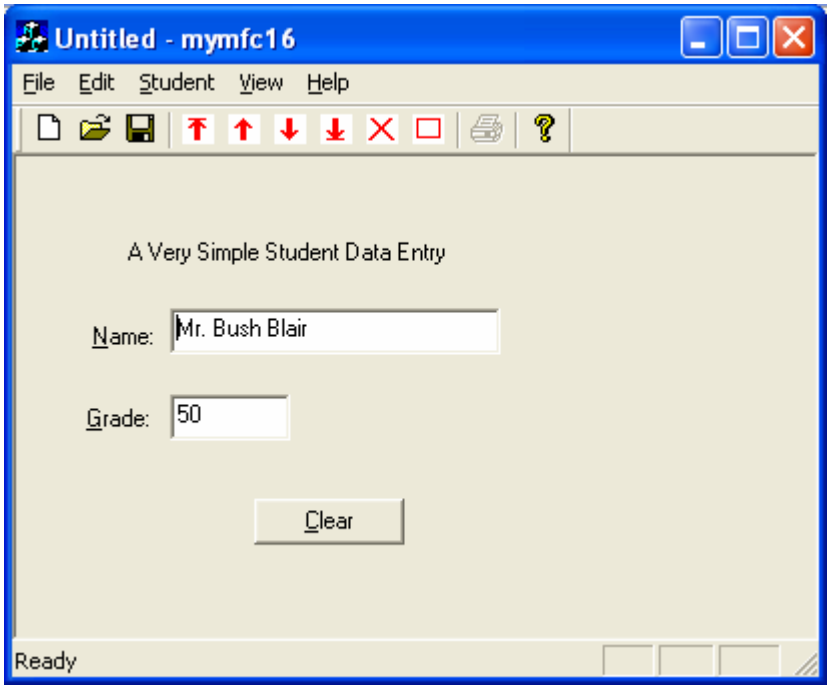


Figure 18: The MYMFC16 program in action.

The toolbar buttons function as follows.

Button	Function
	Retrieves the first student record
	Retrieves the last student record
	Retrieves the previous student record
	Retrieves the next student record
	Deletes the current student record
	Inserts a new student record

Table 6: MYMFC16 new toolbar buttons

The **Clear** button in the view window clears the contents of the **Name** and **Grade** edit controls. The **Clear All** command on the **Edit** menu deletes all the student records in the list and clears the view's edit controls. This example deviates from the step-by-step format in the previous examples. Because there's now more code, we'll simply list selected code and the resource requirements. In the code listing figures, brown color code indicates additional code or other changes that you enter in the output from AppWizard and ClassWizard. The frequent use of TRACE statements lets you follow the program's execution in the debugging window.

Resource Requirements

The file **mymfc16.rc** defines the application's resources as follows.

Toolbar

The toolbar was created by erasing the **Edit Cut**, **Copy**, and **Paste** tiles (fourth, fifth, and sixth from the left) and replacing them with six new patterns.

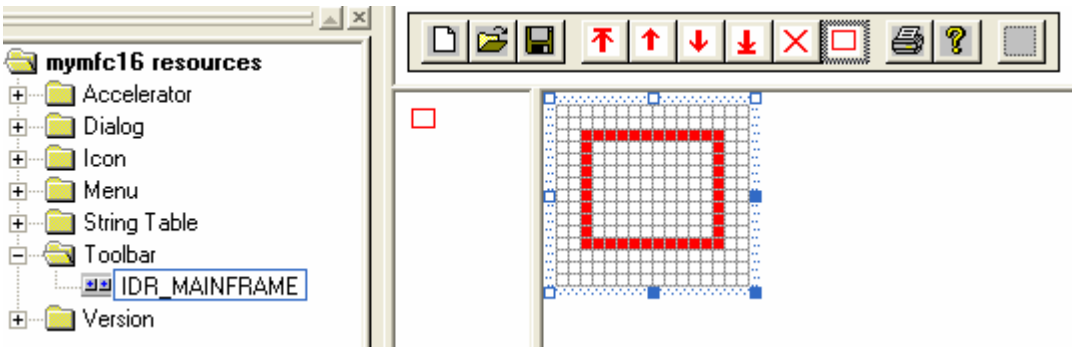


Figure 19: Creating new toolbar buttons for MYMFC16 project.

The **Flip Vertical** command (on the **Image** menu) was used to duplicate some of the tiles.

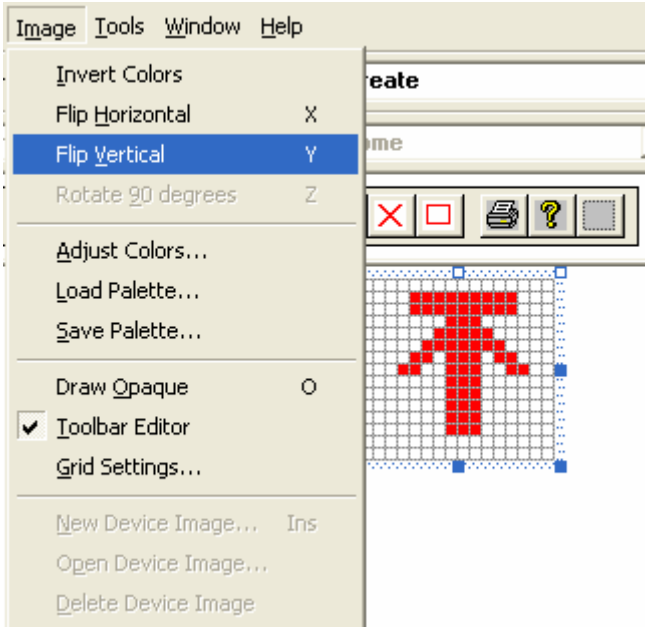


Figure 20: Toolbar's button editor utility under **Image** menu.

The `mymfc16.rc` file defines the linkage between the command IDs and the toolbar buttons.

Student Menu

Having menu options that correspond to the new toolbar buttons isn't absolutely necessary. ClassWizard allows you to map toolbar button commands just as easily as menu commands. However, most applications for Microsoft Windows have menu options for all commands, so users generally expect them.

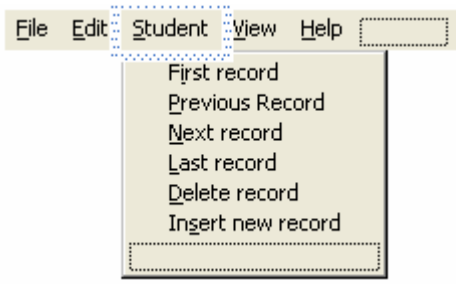


Figure 21: New menu and its items for MYMFC16, in this example just for a completeness.

Edit Menu

On the **Edit** menu, the clipboard menu items are replaced by the **Clear All** menu item. See previous project example for an illustration of the **Edit** menu.

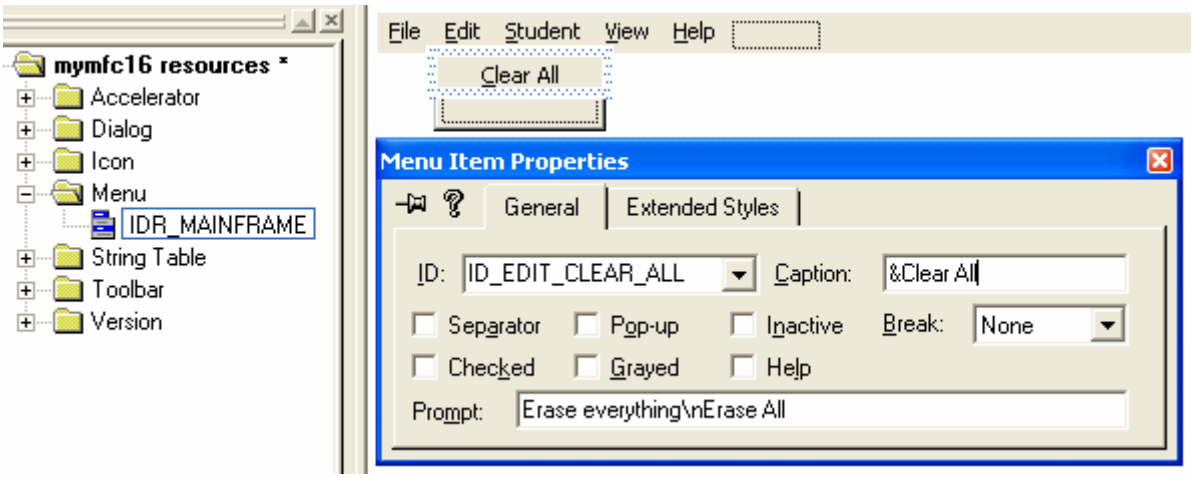


Figure 22: Adding and modifying new **Edit** menu item, **Clear All**.

The `IDD_MYMFC16_FORM` Dialog Template

The `IDD_MYMFC16_FORM` dialog template, shown here, is similar to the MYMFC15 dialog shown in Figure 1 except that the **Enter** pushbutton has been replaced by the **Clear** pushbutton.

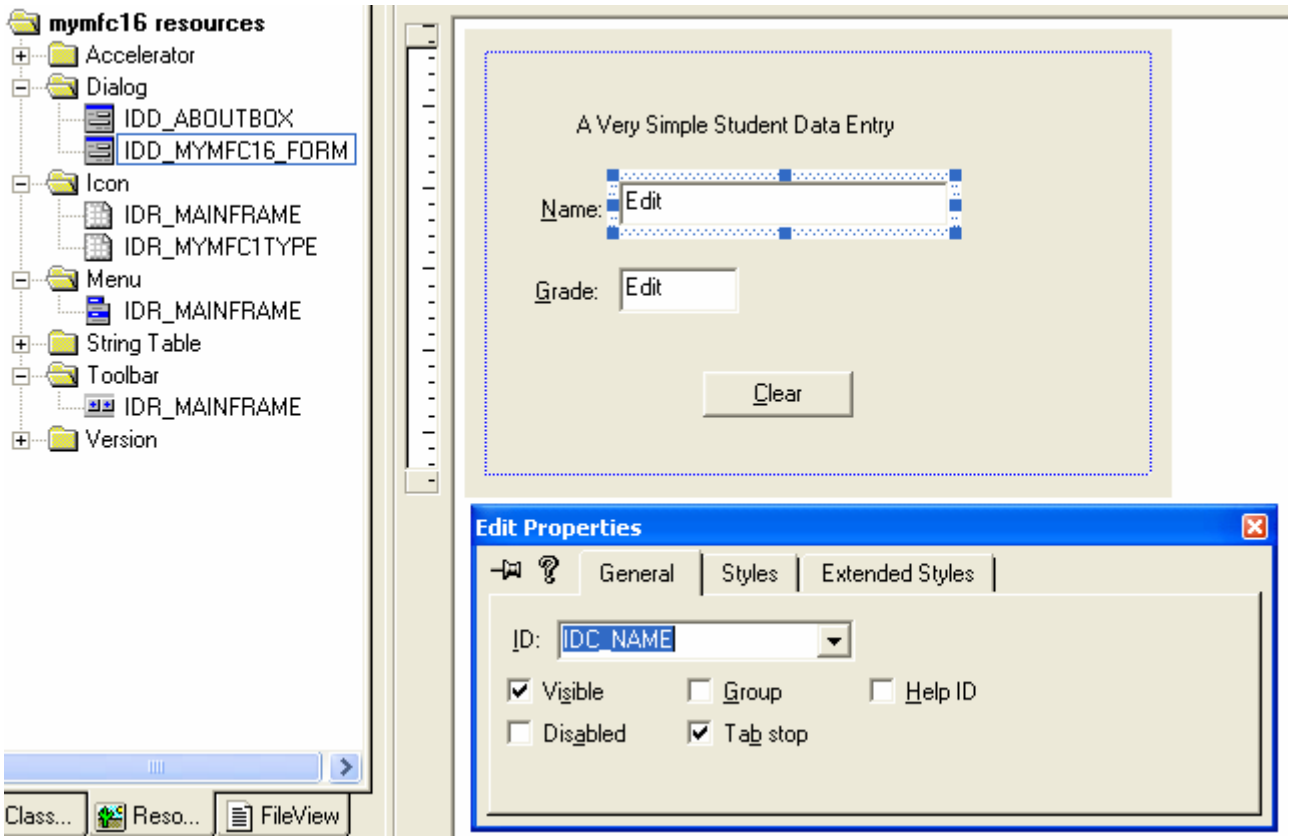


Figure 23: Modifying properties and adding new items to the `IDD_MYMFC16_FORM` dialog.

The following IDs identify the controls.

Control	ID
The dialog template	IDD_MYMFC16_FORM
Name edit control	IDC_NAME
Grade edit control	IDC_GRADE
Clear pushbutton	IDC_CLEAR

Table 7.

The controls' styles are the same as for the MYMFC15 program which the **Styles** properties are: **Style** = **Child**; **Border** = **None** and that Visible is unchecked.

Code Requirements

Here's a list of the files and classes in the MYMFC16 example.

Header File	Source Code File	Classes	Description
mymfc16.h	mymfc16.cpp	CMymfc16App	Application class (from AppWizard)
		CAboutDlg	About dialog
MainFrm.h	MainFrm.cpp	CMainFrame	SDI main frame
mymfc16Doc.h	mymfc16Doc.cpp	CMymfc16Doc	Student document
mymfc16View.h	mymfc16View.cpp	CMymfc16View	Student form view (derived from CFormView)
Student.h	Student.cpp	CStudent	Student record (similar to MYMFC15)
StdAfx.h	StdAfx.cpp	Includes the standard precompiled headers	-

Table 8.

CMymfc16App Class

The files **mymfc16.cpp** and **mymfc16.h** are standard AppWizard output.

CMainFrame Class

The code for the CMainFrame class in **MainFrm.cpp** is standard AppWizard output.

CStudent Class

This is the code from MYMFC15. Insert new header (**Student.h**) and source (**Student.cpp**) files by using the **Project**, **Add To Project** and **New…** menu as shown below. Then, copy the MYMFC15 **Student.h** and **Student.cpp** codes and paste into the MYMFC16 **Student.h** and **Student.cpp** files respectively.

Figure 24: Creating and adding new files, **Student.h** and **Student.cpp** (for new class) to the project.

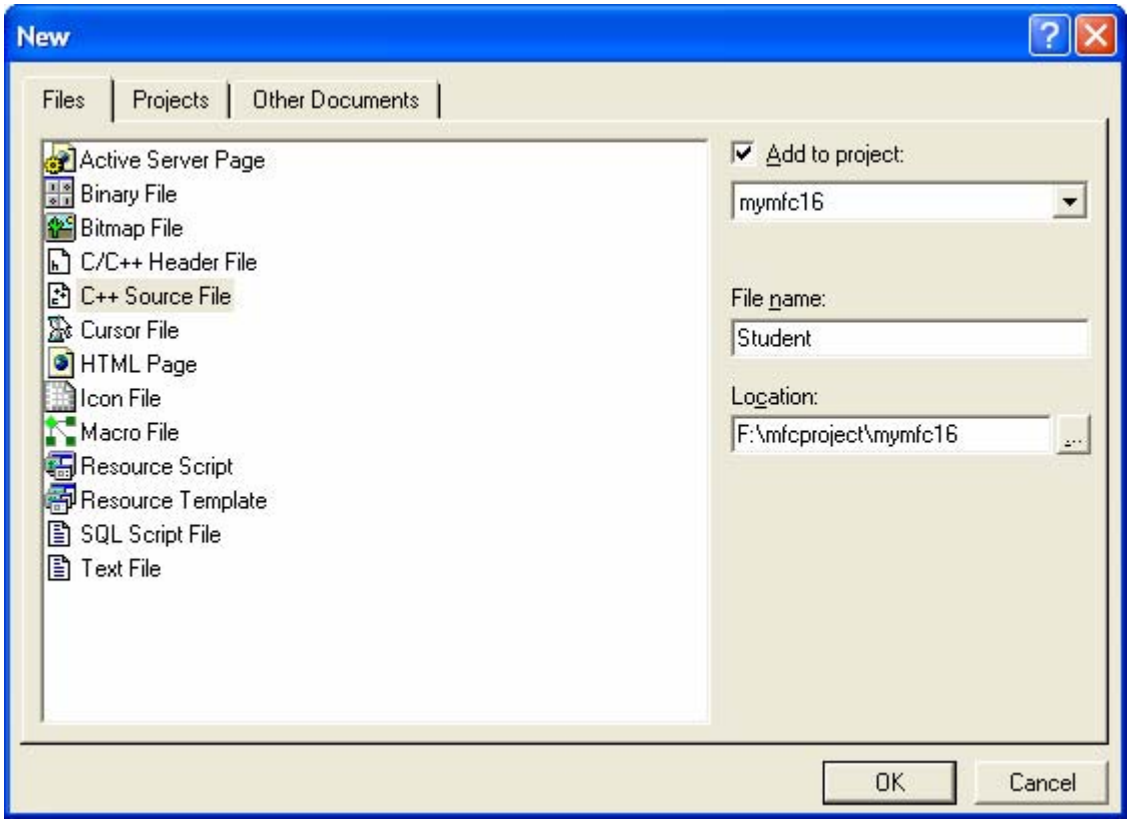


Figure 25: Creating and adding **Student.cpp** file to the project.

Next add the following line at the end of **Student.h**:

```
typedef CTypedPtrList<CObList, CStudent*> CStudentList;
```

```
#ifdef _DEBUG
    void Dump(CDumpContext& dc) const;
#endif // _DEBUG
};

#endif // _INSIDE_VISUAL_CPP_STUDENT

typedef CTypedPtrList<CObList, CStudent*> CStudentList;
```

Listing 11.

The use of the MFC template collection classes requires the following statement in **StdAfx.h**:

```
#include <afxtempl.h>
```

```
#define VC_EXTRALEAN        // Exclu

#include <afxwin.h>         // MFC c
#include <afxext.h>         // MFC e
#include <afxdtctl.h>       // MFC s
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC s
#endif // _AFX_NO_AFXCMN_SUPPORT

#include <afxtempl.h>

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert
```

Listing 12.

ClassWizard and CMymfc16Doc

The **Edit Clear All** command is handled in the document class. The following message handlers were added through ClassWizard.

Object ID	Message	Member Function
ID_EDIT_CLEAR_ALL	COMMAND	OnEditClearAll()
ID_EDIT_CLEAR_ALL	ON_UPDATE_COMMAND_UI	OnUpdateEditClearAll()

Table 9.

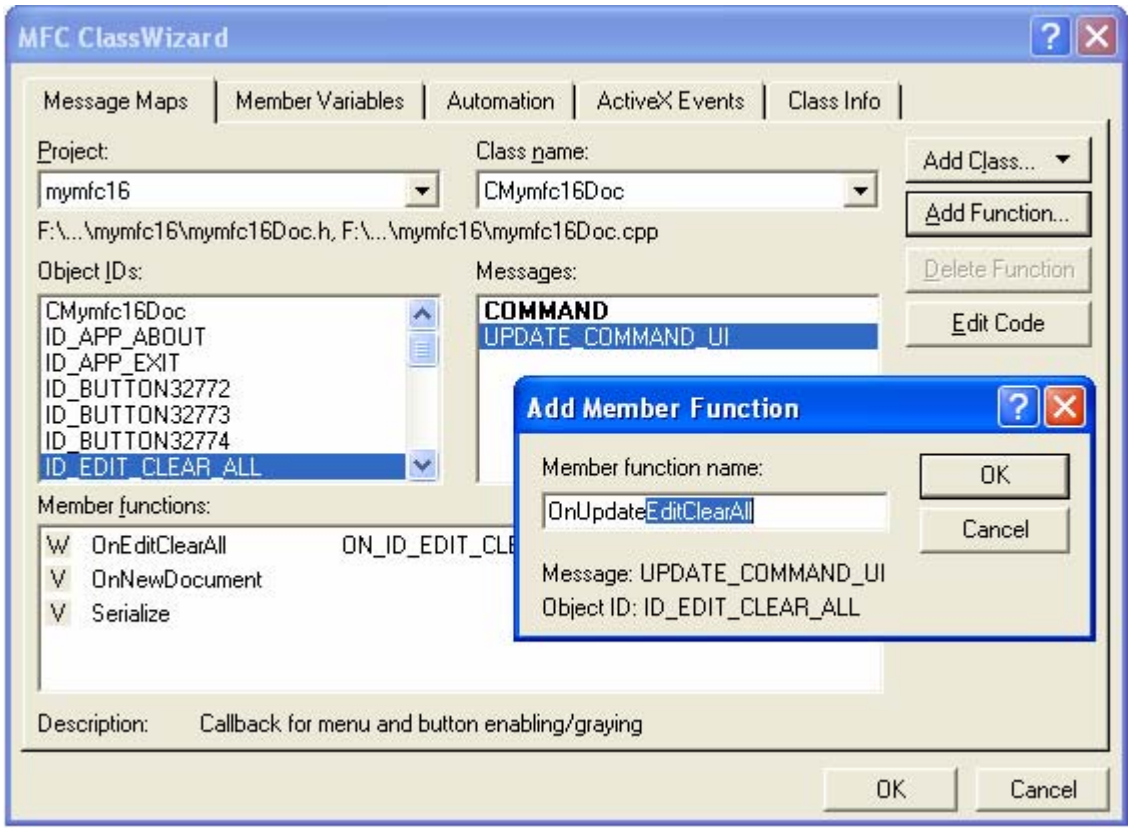


Figure 26: Adding a message handler for the **Edit Clear All** command in the document class.

Data Members

The document class provides for an embedded `CStudentList` object, the `m_studentList` data member, which holds pointers to `CStudent` objects. The list object is constructed when the `CMymfc16Doc` object is constructed, and it is destroyed at program exit. `CStudentList` is a typedef for a `CTypedPtrList` for `CStudent` pointers.

```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CStudentList m_studentList;
};
```

Listing 13.

Constructor

The document constructor sets the depth of the dump context so that a dump of the list causes dumps of the individual list elements.

```
CMymfc16Doc::CMymfc16Doc()
{
    TRACE("Entering CMymfc16Doc constructor\n");
#ifdef _DEBUG
    afxDump.SetDepth(1); // Ensure dump of list elements
#endif // _DEBUG
}
```

Listing 14.

GetList()

The inline `GetList()` function helps isolate the view from the document. The document class must be specific to the type of object in the list, in this case, objects of the class `CStudent()`. A generic list view base class, however, can use a member function to get a pointer to the list without knowing the name of the list object.

```
// Attributes
public:
    CStudentList* GetList() {
        return &m_studentList;
    }
```

Listing 15.

DeleteContents()

The `DeleteContents()` function is a virtual override function that is called by other document functions and by the application framework. Its job is to remove all student object pointers from the document's list and to delete those student objects. An important point to remember here is that SDI document objects are reused after they are closed. `DeleteContents()` also dumps the student list.

```
void CMymfc16Doc::DeleteContents()
{
#ifdef _DEBUG
    Dump(afxDump);
#endif
    while (m_studentList.GetHeadPosition())
    {
        delete m_studentList.RemoveHead();
    }
}
```

Listing 16.

`Dump()`

AppWizard generates the `Dump()` function skeleton between the lines `#ifdef _DEBUG` and `#endif`. Because the `afxDump` depth was set to 1 in the document constructor, all the `CStudent` objects contained in the list are dumped.

```
void CMymfc16Doc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
    dc << "\n" << m_studentList << "\n";
}
#endif //_DEBUG
```

Listing 17.

`CMymfc16Doc` Class

AppWizard originally generated the `CMymfc16Doc` class. Listing 18 shows the code used in the MYMFC16 example.

MYMFC16DOC.H

```
// Mymfc16Doc.h : interface of the CMymfc16Doc class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_MYMFC16DOC_H__4D011047_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
#define AFX_MYMFC16DOC_H__4D011047_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "student.h"

class CMymfc16Doc : public CDocument
{
protected: // create from serialization only
    CMymfc16Doc();
    DECLARE_DYNCREATE(CMymfc16Doc)

// Attributes
public:
    CStudentList* GetList() {
        return &m_studentList;
    }

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
```

```
    //{{AFX_VIRTUAL(CMymfc16Doc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual void DeleteContents();
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMymfc16Doc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CMymfc16Doc)
    afx_msg void OnEditClearAll();
    afx_msg void OnUpdateEditClearAll(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CStudentList m_studentList;
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.

#endif //
!defined(AFX_MYMFC16DOC_H__4D011047_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
```

MYMFC16DOC.CPP

```
// Mymfc16Doc.cpp : implementation of the CMymfc16Doc class
//

#include "stdafx.h"
#include "mymfc16.h"

#include "Mymfc16Doc.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMymfc16Doc

IMPLEMENT_DYNCREATE(CMymfc16Doc, CDocument)

BEGIN_MESSAGE_MAP(CMymfc16Doc, CDocument)
    //{{AFX_MSG_MAP(CMymfc16Doc)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
    ON_UPDATE_COMMAND_UI(ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMymfc16Doc construction/destruction
```

```
CMymfc16Doc::CMymfc16Doc()
{
    TRACE("Entering CMymfc16Doc constructor\n");
#ifdef _DEBUG
    afxDump.SetDepth(1); // Ensure dump of list elements
#endif // _DEBUG
}

CMymfc16Doc::~CMymfc16Doc()
{
}

BOOL CMymfc16Doc::OnNewDocument()
{
    TRACE("Entering CMymfc16Doc::OnNewDocument\n");
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add re-initialization code here
    // (SDI documents will reuse this document)

    return TRUE;
}
/////////////////////////////////////////////////////////////////////////////
// CMymfc16Doc serialization

void CMymfc16Doc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}

/////////////////////////////////////////////////////////////////////////////
// CMymfc16Doc diagnostics

#ifdef _DEBUG
void CMymfc16Doc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMymfc16Doc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
    dc << "\n" << m_studentList << "\n";
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMymfc16Doc commands

void CMymfc16Doc::DeleteContents()
{
#ifdef _DEBUG
    Dump(afxDump);
#endif
    while (m_studentList.GetHeadPosition()) {
        delete m_studentList.RemoveHead();
    }
```

```
}

void CMymfc16Doc::OnEditClearAll()
{
    DeleteContents();
    UpdateAllViews(NULL);
}

void CMymfc16Doc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!m_studentList.IsEmpty());
}
```

Listing 18: The `CMymfc16Doc` class listing.

ClassWizard and `CMymfc16View` Class

ClassWizard was used to map the `CMymfc16View` **Clear** pushbutton notification message as follows.

Object ID	Message	Member Function
IDC_CLEAR	BN_CLICKED	OnClear()

Table 10.

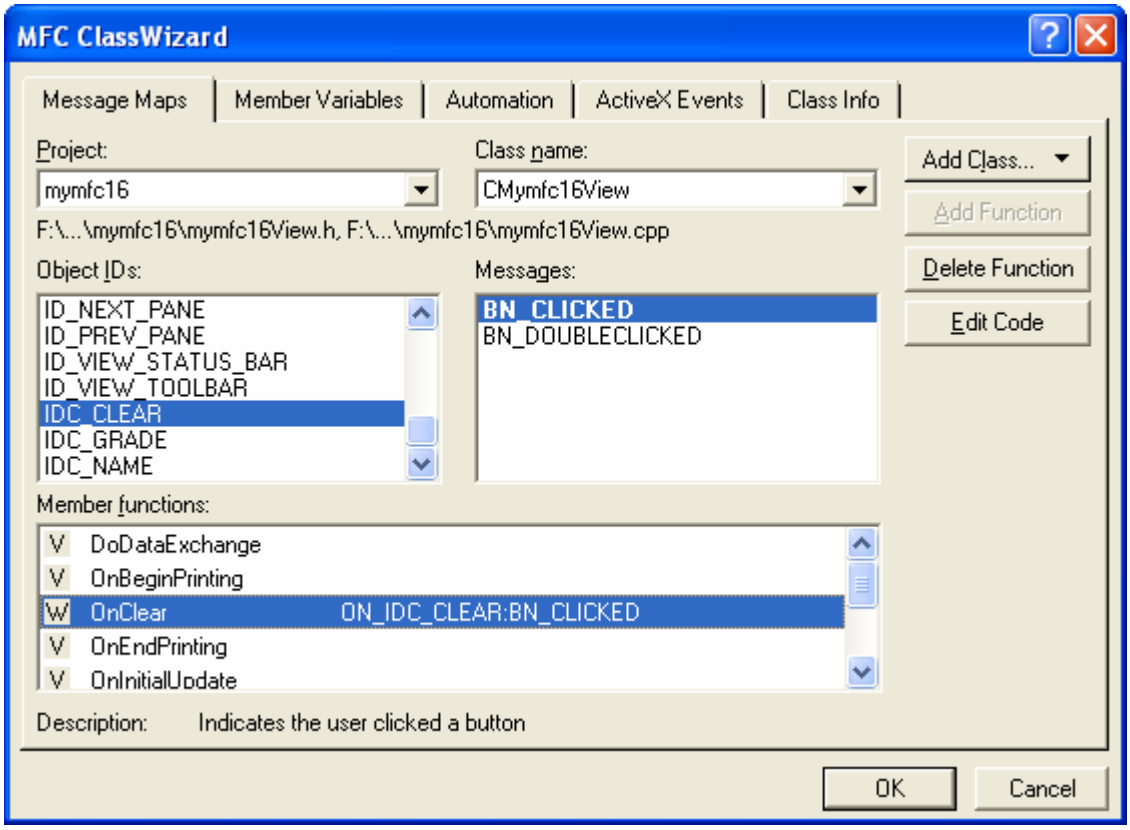


Figure 27: Mapping the `CMymfc16View` **Clear** pushbutton notification message.

Because `CMymfc16View` is derived from `CFormView`, ClassWizard supports the definition of dialog data members. The variables shown here were added with the **Add Variables** button.

Control ID	Member Variable	Category	Variable Type
IDC_GRADE	m_nGrade	Value	int

IDC_NAME	m_strName	Value	CString

Table 11.

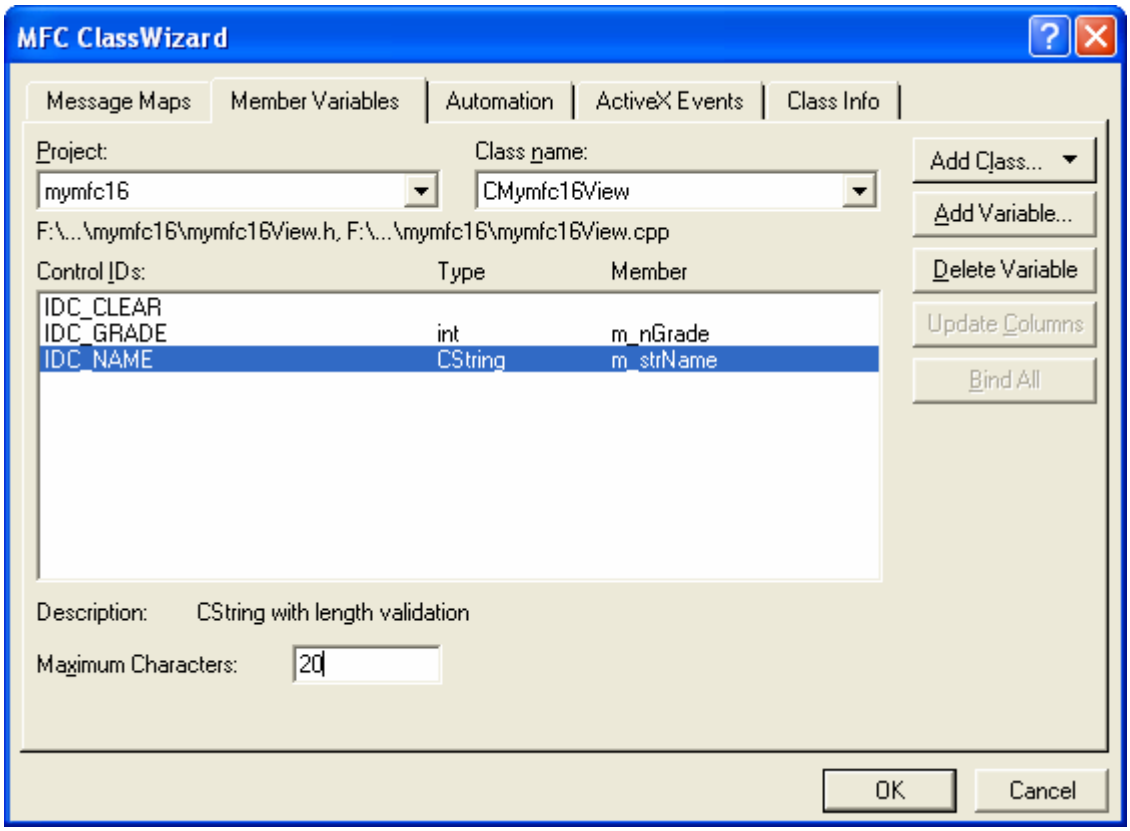


Figure 28: Adding member variables.

The minimum value of the `m_nGrade` data member was set to **0**, and its maximum value was set to **100**. The maximum length of the `m_strName` data member was set to **20** characters.

ClassWizard maps toolbar button commands to their handlers.

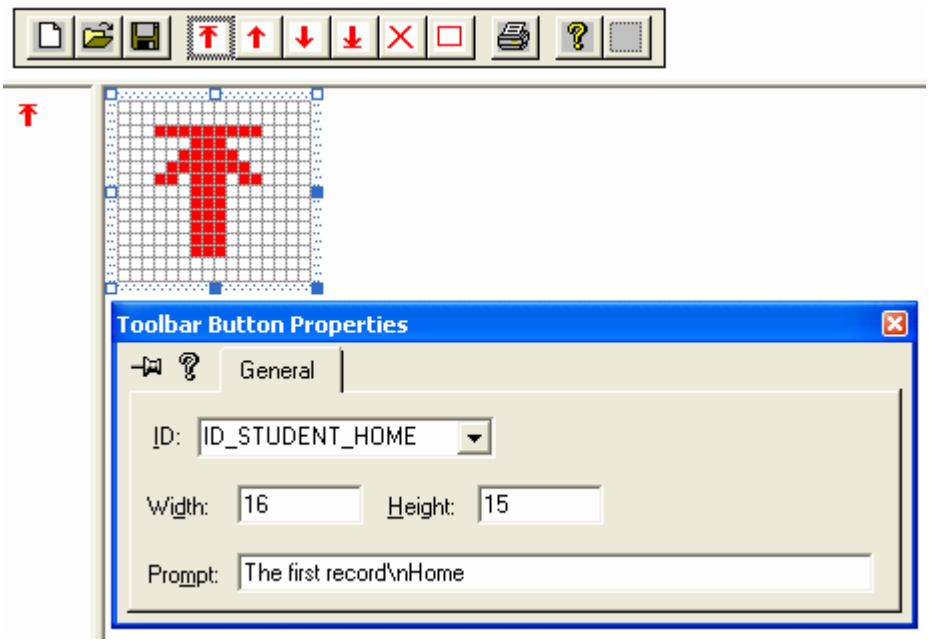


Figure 29: Modifying the toolbar button properties.

Here are the commands and the handler functions to which they were mapped.

	Object ID	Message	Member Function
	ID_STUDENT_HOME	COMMAND	OnStudentHome()
	ID_STUDENT_END	COMMAND	OnStudentEnd()
	ID_STUDENT_PREV	COMMAND	OnStudentPrev()
	ID_STUDENT_NEXT	COMMAND	OnStudentNext()
	ID_STUDENT_INS	COMMAND	OnStudentIns()
	ID_STUDENT_DEL	COMMAND	OnStudentDel()

Table 12.

The message mapping using ClassWizard.

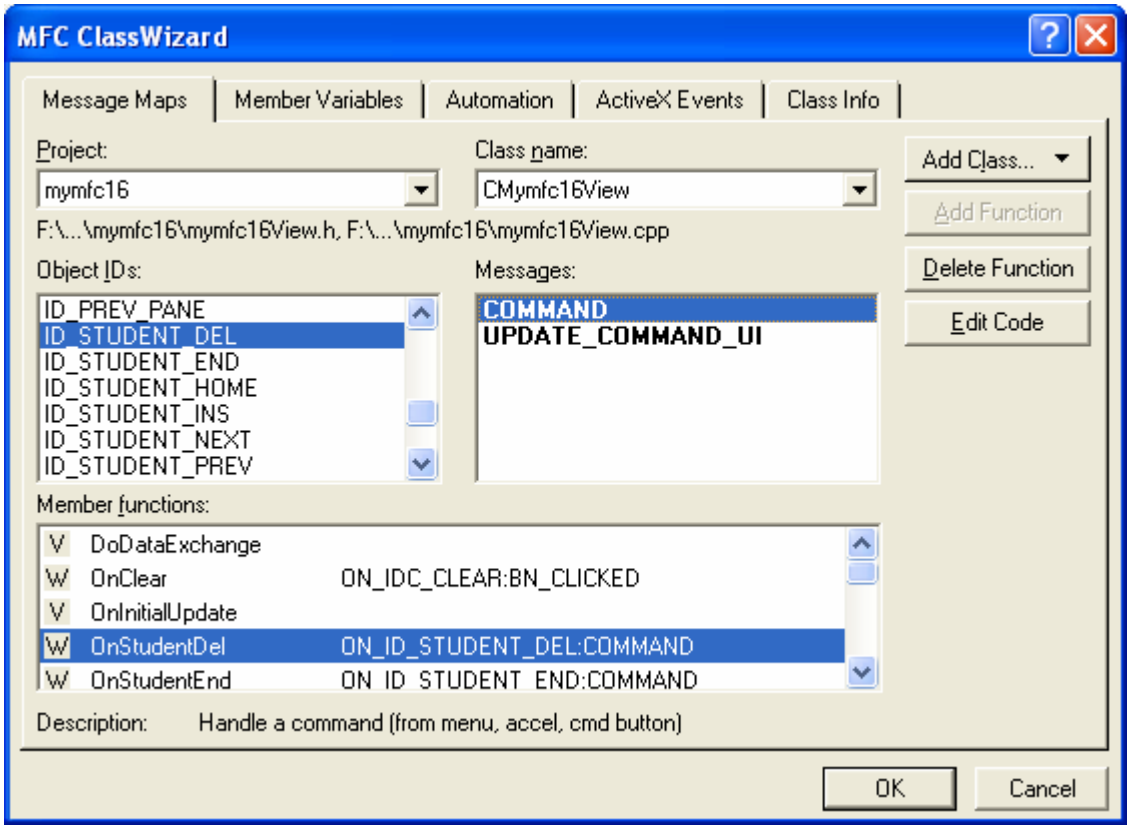


Figure 30: Message mapping of the commands and their handler functions.

Each command handler has built-in error checking. The following update command UI message handlers are called during idle processing to update the state of the toolbar buttons and when the **Student** menu is painted, to update the menu items.

Object ID	Message	Member Function
ID_STUDENT_HOME	UPDATE_COMMAND_UI	OnUpdateStudentHome()
ID_STUDENT_END	UPDATE_COMMAND_UI	OnUpdateStudentEnd()
ID_STUDENT_PREV	UPDATE_COMMAND_UI	OnUpdateStudentHome()
ID_STUDENT_NEXT	UPDATE_COMMAND_UI	OnUpdateStudentEnd()
ID_STUDENT_DEL	UPDATE_COMMAND_UI	OnUpdateCommandDel()

Table 13.

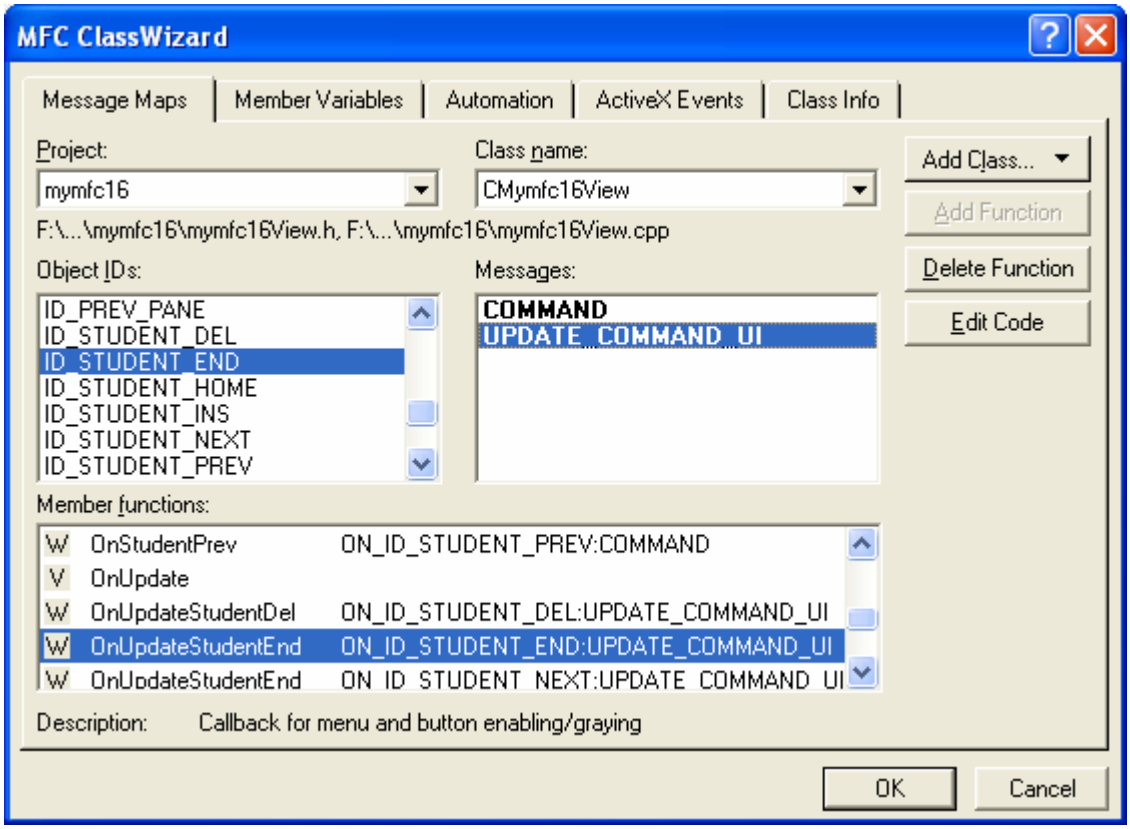


Figure 31: Message mapping of the commands and their UI handler functions.

For example, this button: 

Which retrieves the first student record, is disabled when the list is empty and when the `m_position` variable is already set to the head of the list. The **Previous** button is disabled under the same circumstances, so it uses the same update command UI handler. The **End** and the **Next** buttons share a handler for similar reasons. Because a delay sometimes occurs in calling the update command UI functions, the command message handlers must look for error conditions.

Data Members

The `m_position` data member is a kind of cursor for the document's collection. It contains the position of the `CStudent` object that is currently displayed. The `m_pList` variable provides a quick way to get at the student list in the document.

```
class CMymfc16View : public CFormView
{
protected:
    POSITION      m_position; // current
    CStudentList* m_pList;    // copied
```

Listing 19.

OnInitialUpdate()

The virtual `OnInitialUpdate()` function is called when you start the application. It sets the view's `m_pList` data member for subsequent access to the document's list object.

```
void CMymfc16View::OnInitialUpdate()
{
    TRACE("Entering CMymfc16View::OnInitialUpdate\n");
    m_pList = GetDocument()->GetList();
    CFormView::OnInitialUpdate();
}
```

Listing 20.

`OnUpdate()`

The virtual `OnUpdate()` function is called both by the `OnInitialUpdate()` function and by the `CDocument::UpdateAllViews` function. It resets the list position to the head of the list, and it displays the head entry. In this example, the `UpdateAllViews()` function is called only in response to the **Edit Clear All** command. In a multiview application, you might need a different strategy for setting the `CMymfc16View` `m_position` variable in response to document updates from another view.

```
void CMymfc16View::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    // called by OnInitialUpdate and by UpdateAllViews
    TRACE("Entering CMymfc16View::OnUpdate\n");
    m_position = m_pList->GetHeadPosition();
    GetEntry(m_position); // initial data for view
}
```

Listing 21.

Protected Virtual Functions

The following three functions are protected virtual functions that deal specifically with `CStudent` objects:

1. `GetEntry()`.
2. `InsertEntry()`.
3. `ClearEntry()`.

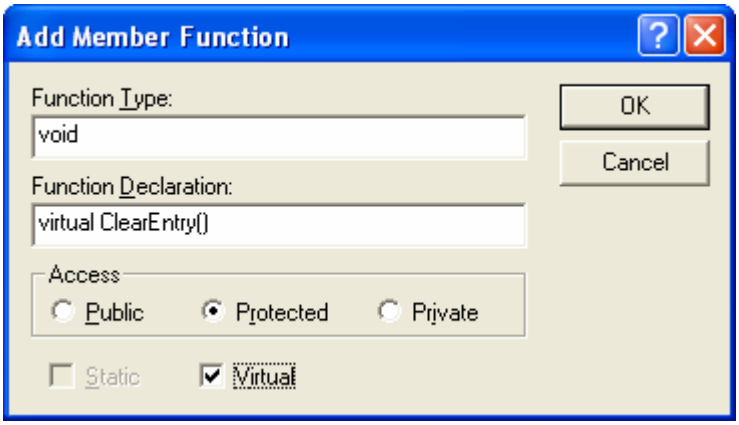


Figure 32: Adding a protected virtual function using ClassView.

```
void CMymfc16View::GetEntry(POSITION position)
{
    if (position) {
        CStudent* pStudent = m_pList->GetAt(position);
        m_strName = pStudent->m_strName;
        m_nGrade = pStudent->m_nGrade;
    }
    else {
        ClearEntry();
    }
    UpdateData(FALSE);
}

void CMymfc16View::InsertEntry(POSITION position)
{
    if (UpdateData(TRUE)) {
        // UpdateData returns FALSE if it detects a user error
        CStudent* pStudent = new CStudent;
        pStudent->m_strName = m_strName;
        pStudent->m_nGrade = m_nGrade;
        m_position = m_pList->InsertAfter(m_position, pStudent);
    }
}

void CMymfc16View::ClearEntry()
{
    m_strName = "";
    m_nGrade = 0;
    UpdateData(FALSE);
    ((CDialog*) this)->GotoDlgCtrl(GetDlgItem(IDC_NAME));
}
```

Listing 22.

You can transfer these functions to a derived class if you want to isolate the general-purpose list-handling features in a base class.

`CMymfc16View` Class

Listing 23 shows the code for the `CMymfc16View` class. This code will be carried over into the next two Modules.

MYMFC16VIEW.H

```
// Mymfc16View.h : interface of the CMymfc16View class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_MYMFC16VIEW_H__4D011049_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
#define AFX_MYMFC16VIEW_H__4D011049_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CMymfc16View : public CFormView
{
protected:
    POSITION      m_position; // current position in document list
    CStudentList* m_pList;    // copied from document

protected: // create from serialization only
    CMymfc16View();
    DECLARE_DYNCREATE(CMymfc16View)

public:
```

```
    //{{AFX_DATA(CMymfc16View)
    enum { IDD = IDD_MYMFC16_FORM };
    int     m_nGrade;
    CString m_strName;
    //}}AFX_DATA

// Attributes
public:
    CMymfc16Doc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMymfc16View)
    public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    virtual void OnInitialUpdate(); // called first time after construct
    virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMymfc16View();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:
    virtual void ClearEntry();
    virtual void InsertEntry(POSITION position);
    virtual void GetEntry(POSITION position);

// Generated message map functions
protected:
    //{{AFX_MSG(CMymfc16View)
    afx_msg void OnClear();
    afx_msg void OnStudentHome();
    afx_msg void OnStudentEnd();
    afx_msg void OnStudentPrev();
    afx_msg void OnStudentNext();
    afx_msg void OnStudentIns();
    afx_msg void OnStudentDel();
    afx_msg void OnUpdateStudentHome(CCmdUI* pCmdUI);
    afx_msg void OnUpdateStudentEnd(CCmdUI* pCmdUI);
    afx_msg void OnUpdateStudentDel(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in Mymfc16View.cpp
inline CMymfc16Doc* CMymfc16View::GetDocument()
   { return (CMymfc16Doc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.

#endif //
```

```
!defined(AFX_MYMFC16VIEW_H__4D011049_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
```

MYMFC16VIEW.CPP

```
// Mymfc16View.cpp : implementation of the CMymfc16View class
//

#include "stdafx.h"
#include "mymfc16.h"

#include "Mymfc16Doc.h"
#include "Mymfc16View.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMymfc16View

IMPLEMENT_DYNCREATE(CMymfc16View, CFormView)
BEGIN_MESSAGE_MAP(CMymfc16View, CFormView)
    //{{AFX_MSG_MAP(CMymfc16View)
    ON_BN_CLICKED(IDC_CLEAR, OnClear)
    ON_COMMAND(ID_STUDENT_HOME, OnStudentHome)
    ON_COMMAND(ID_STUDENT_END, OnStudentEnd)
    ON_COMMAND(ID_STUDENT_PREV, OnStudentPrev)
    ON_COMMAND(ID_STUDENT_NEXT, OnStudentNext)
    ON_COMMAND(ID_STUDENT_INS, OnStudentIns)
    ON_COMMAND(ID_STUDENT_DEL, OnStudentDel)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_HOME, OnUpdateStudentHome)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_END, OnUpdateStudentEnd)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_PREV, OnUpdateStudentHome)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_NEXT, OnUpdateStudentEnd)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_DEL, OnUpdateStudentDel)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMymfc16View construction/destruction

CMymfc16View::CMymfc16View() : CFormView(CMymfc16View::IDD)
{
    TRACE("Entering CMymfc16View constructor\n");
    //{{AFX_DATA_INIT(CMymfc16View)
    m_nGrade = 0;
    m_strName = _T("");
    //}}AFX_DATA_INIT
    m_position = NULL;
}

CMymfc16View::~CMymfc16View()
{
}

void CMymfc16View::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CMymfc16View)
    DDX_Text(pDX, IDC_GRADE, m_nGrade);
    DDV_MinMaxInt(pDX, m_nGrade, 0, 100);
    DDX_Text(pDX, IDC_NAME, m_strName);
    DDV_MaxChars(pDX, m_strName, 20);
    //}}AFX_DATA_MAP
}
```

```
BOOL CMymfc16View::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs
    return CFormView::PreCreateWindow(cs);
}

void CMymfc16View::OnInitialUpdate()
{
    TRACE("Entering CMymfc16View::OnInitialUpdate\n");
    m_pList = GetDocument()->GetList();
    CFormView::OnInitialUpdate();
}

/////////////////////////////////////////////////////////////////////////
// CMymfc16View diagnostics

#ifdef _DEBUG
void CMymfc16View::AssertValid() const
{
    CFormView::AssertValid();
}

void CMymfc16View::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CMymfc16Doc* CMymfc16View::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMymfc16Doc)));
    return (CMymfc16Doc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////
// CMymfc16View message handlers

void CMymfc16View::OnClear()
{
    TRACE("Entering CMymfc16View::OnClear\n");
    ClearEntry();
}

void CMymfc16View::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    // called by OnInitialUpdate and by UpdateAllViews
    TRACE("Entering CMymfc16View::OnUpdate\n");
    m_position = m_pList->GetHeadPosition();
    GetEntry(m_position); // initial data for view
}

void CMymfc16View::OnStudentHome()
{
    TRACE("Entering CMymfc16View::OnStudentHome\n");
    // need to deal with list empty condition
    if (!m_pList->IsEmpty()) {
        m_position = m_pList->GetHeadPosition();
        GetEntry(m_position);
    }
}

void CMymfc16View::OnStudentEnd()
{
    TRACE("Entering CMymfc16View::OnStudentEnd\n");
    if (!m_pList->IsEmpty()) {
```

```
        m_position = m_pList->GetTailPosition();
        GetEntry(m_position);
    }
}

void CMymfc16View::OnStudentPrev()
{
    POSITION pos;
    TRACE("Entering CMymfc16View::OnStudentPrev\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetPrev(pos);
        if (pos) {
            GetEntry(pos);
      m_position = pos;
        }
    }
}

void CMymfc16View::OnStudentNext()
{
    POSITION pos;
    TRACE("Entering CMymfc16View::OnStudentNext\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetNext(pos);
        if (pos) {
            GetEntry(pos);
            m_position = pos;
        }
    }
}

void CMymfc16View::OnStudentIns()
{
    TRACE("Entering CMymfc16View::OnStudentIns\n");
    InsertEntry(m_position);
    GetDocument()->SetModifiedFlag();
    GetDocument()->UpdateAllViews(this);
}

void CMymfc16View::OnStudentDel()
{
    // deletes current entry and positions to next one or head
    POSITION pos;
    TRACE("Entering CMymfc16View::OnStudentDel\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetNext(pos);
        if (pos == NULL) {
            pos = m_pList->GetHeadPosition();
            TRACE("GetHeadPos = %ld\n", pos);
            if (pos == m_position) {
                pos = NULL;
            }
        }
        GetEntry(pos);
        CStudent* ps = m_pList->GetAt(m_position);
        m_pList->RemoveAt(m_position);
        delete ps;
        m_position = pos;
        GetDocument()->SetModifiedFlag();
        GetDocument()->UpdateAllViews(this);
    }
}

void CMymfc16View::OnUpdateStudentHome(CCmdUI* pCmdUI)
{
    // called during idle processing and when Student menu drops down
```

```
    POSITION pos;

    // enables button if list not empty and not at home already
    pos = m_pList->GetHeadPosition();
    pCmdUI->Enable((m_position != NULL) && (pos != m_position));
}

void CMymfc16View::OnUpdateStudentEnd(CCmdUI* pCmdUI)
{
    // called during idle processing and when Student menu drops down
    POSITION pos;

    // enables button if list not empty and not at end already
    pos = m_pList->GetTailPosition();
    pCmdUI->Enable((m_position != NULL) && (pos != m_position));
}

void CMymfc16View::OnUpdateStudentDel(CCmdUI* pCmdUI)
{
    // called during idle processing and when Student menu drops down
    pCmdUI->Enable(m_position != NULL);
}

void CMymfc16View::GetEntry(POSITION position)
{
    if (position) {
        CStudent* pStudent = m_pList->GetAt(position);
        m_strName = pStudent->m_strName;
        m_nGrade = pStudent->m_nGrade;
    }
    else {
        ClearEntry();
    }
    UpdateData(FALSE);
}

void CMymfc16View::InsertEntry(POSITION position)
{
    if (UpdateData(TRUE)) {
        // UpdateData returns FALSE if it detects a user error
        CStudent* pStudent = new CStudent;
        pStudent->m_strName = m_strName;
        pStudent->m_nGrade = m_nGrade;
        m_position = m_pList->InsertAfter(m_position, pStudent);
    }
}

void CMymfc16View::ClearEntry()
{
    m_strName = "";
    m_nGrade = 0;
    UpdateData(FALSE);
    ((CDialog*) this)->GotoDlgCtrl(GetDlgItem(IDC_NAME));
}
```

Listing 23: The `CMymfc16View` class listing.

Testing the MYMFC16 Application

Build the program and start it from the debugger.

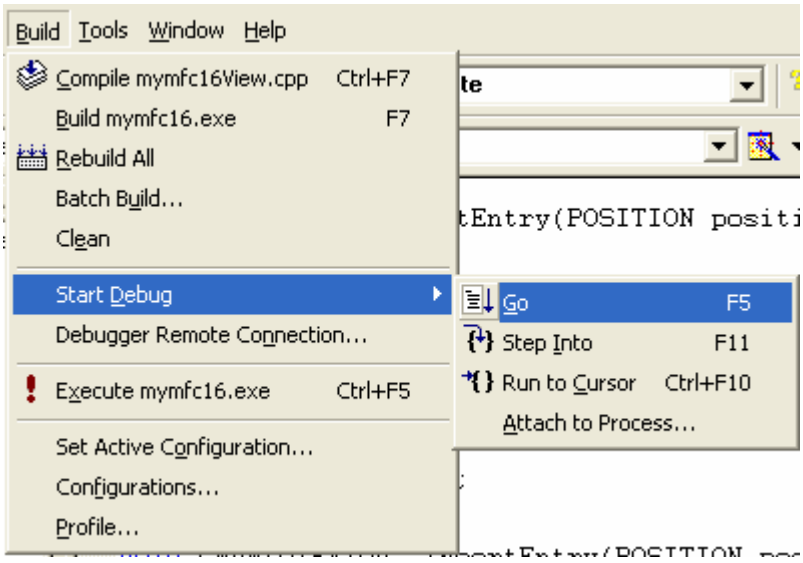


Figure 33: Running MYMFC16 program from the debugger.

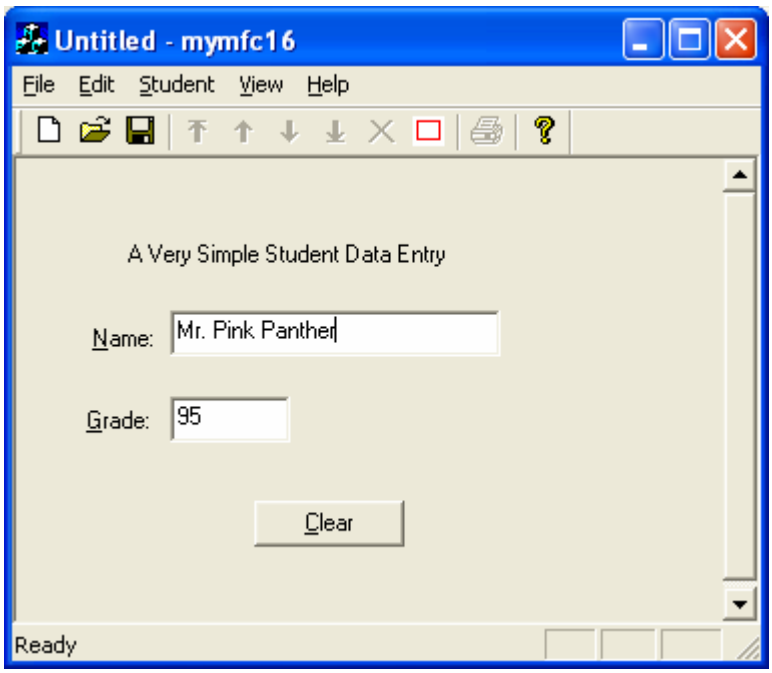


Figure 34: MYMFC16 program output in action.

Fill in the **student name** and **grade fields**, and then click the **New** button:

To insert the entry into the list. Repeat this action several times, using the **Clear** pushbutton to erase the data from the previous entry. Notice the toolbar buttons enable/disable change. When you exit the application, the debug output should look similar to this:

```
...
Entering CMymfc16View::OnStudentIns
Entering CMymfc16View::OnClear
Entering CMymfc16View::OnStudentIns
Entering CMymfc16View::OnClear
```

```
Entering CMymfc16View::OnStudentIns
Entering CMymfc16View::OnClear
Entering CMymfc16View::OnStudentIns
Entering CMymfc16View::OnClear
a CMymfc16Doc at $4216B0
m_strTitle = Untitled
m_strPathName =
m_bModified = 1
m_pDocTemplate = $4218C0

a CObList at $421704
with 4 elements
       a CStudent at $422DD0
m_strName = Mr. Pink Panther
m_nGrade = 95
       a CStudent at $422340
m_strName = Mr. Bush Blair
m_nGrade = 40
       a CStudent at $422460
m_strName = Mrs. Rice Plate
m_nGrade = 67
       a CStudent at $422200
m_strName = Mr. Mechanick
m_nGrade = 88

Warning: destroying an unsaved document.
The thread 0x5B8 has exited with code 0 (0x0).
The program 'F:\mfcproject\mymfc16\Debug\mymfc16.exe' has exited with code 0 (0x0).
```

You can clear all data by using the **Edit**, **Clear All** menu.

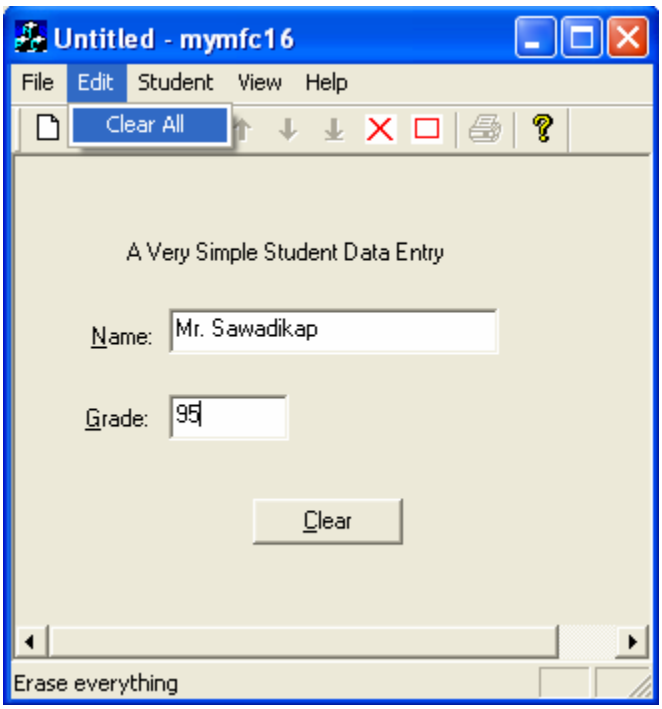


Figure 35: Deleting all the data using the **Edit Clear All** menu.

Two Exercises for the Reader

You might have noticed the absence of a **Modify** button on the toolbar. Without such a button, you can't modify an existing student record. Can you add the necessary toolbar button and message handlers? The most difficult task might be designing a graphic for the button's tile.

Recall that the `CMymfc16View` class is just about ready to be a general-purpose base class. Try separating the `CStudent`-specific virtual functions into a derived class. After that, make another derived class that uses a new element class other than `CStudent`.

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.