Chapter 5

Developing ODBC Projects

IN THIS CHAPTER

- Easily developing a database application by using the MFC Application Wizard
- Adding, deleting, and querying records in an MFC Application Wizard application
- Processing transactions inside an MFC/ODBC application
- Easily joining tables with the MFC Application Wizard
- Opening multiple recordsets within a single application

DATABASE DEVELOPMENT can be extremely complicated. You need to know not only the programming language (Visual C++) but also the database language (SQL), the database setup, and the database API. Visual C++ simplifies this process by making database development somewhat easier.

Some of the function calls may seem cryptic, especially if you don't read this book sequentially. The preceding chapter contained an overview of ODBC with MFC and Visual C++ and showed what several of the functions can do.

Easy Development with an ODBC Application

It really is easy to develop a fully functional Visual C++ database application. To develop an application that scrolls through a table and enables you to update any record takes only ten easy steps:

1. Click File → New to open the New dialog box as seen in Figure 5-1. Click the Projects tab, then choose MFC AppWizard (exe). Type in a project name (ODBCDepartment for this example), and click OK.
Figure 5-1: The New dialog box enables you to choose which kind of project you wish to create. Here, an MFC Application Wizard executable is chosen.

2. The MFC App Wizard - Step 1 of 6 dialog box now opens. Click Single Document and Document/View architecture support. Then click Next.

3. In the MFC App Wizard - Step 2 of 6 dialog box, choose Database view without file support. Then click the Data Source button as shown in Figure 5-2.

Figure 5-2: The MFC Application Wizard enables you to assign a data source to your MFC application.

4. In the Database Options dialog box, choose ODBC and Dynaset. Then choose your ODBC data source. In Figure 5-3, the Classes ODBC data source was chosen.
5. Next, the Select Database Tables dialog box opens (Figure 5-4). After you have chosen your data source, you now must pick the table or tables that you want to include inside your MFC application. In Figure 5-4, the Department table is chosen. After you click OK, the MFC App Wizard - Step 2 of 6 dialog box reappears. You should see your database and table listed below the Data Source button. Click Next.

6. The MFC App Wizard - Step 3 of 6 dialog box opens. Accept all the defaults, and click Next. The MFC App Wizard - Step 4 of 6 dialog box then opens. Deselect (turn off) Printing and print preview, and click Next. The MFC App Wizard - Step 5 of 6 dialog box then opens. Accept all the defaults, and click Next.

7. The MFC App Wizard - Step 6 of 6 dialog box opens. Here, you can change the program names of the C++ classes that will be generated for you. Usually you accept all the defaults, and click Finish.
Here, you should see the New Project Information message box appear. If you scroll down in the box a little, your options should look like the options in Figure 5-5. After you click OK, your project is created, and a dialog box is opened. You can use the painting tools to paint your dialog box.

![New Project Information](image)

Figure 5-5: The New Project Information message box describes all the options that you've chosen for your MFC application.

8. On the dialog box that opens, delete the TODO label. Then add any fields you need to contain your database fields. In Figure 5-6, two edit boxes were added to contain the two columns in the department table. Then the properties were opened, and the edit box names were changed to `IDC_DEPARTMENTCODE` and `IDC_DEPARTMENTNAME`.

Be sure you rename all your window controls to names that correspond to your database column names. Development is easier when you know which controls need to be tied to which field.

9. Pull up the MFC Class Wizard by clicking View → Class Wizard. Click the Member Variables tab. In Figure 5-7, you can see the names of the control IDs that you just added to your dialog box.
Figure 5-6: Add controls for each database column, and use the Properties window to rename each control to a more appropriate name.

Figure 5-7: The Member Variables tab of the MFC Class Wizard lists all the window controls that you’ve added to your MFC application.
10. Double-click each control listed in the Member Variables tab of the MFC Class Wizard. This enables you to pull up the Add Member Variable dialog box. Here, you can choose which database column is to be bound to which window control. In Figure 5-8, the `m_DepartmentCode` variable contained in the `m_pSet CRecordset` variable is joined to the `IDC_DEPARTMENTCODE` window control.

![Add Member Variable dialog box](image)

Figure 5-8: The Add Member Variable dialog box enables you to easily bind the variables of your `CRecordset` class to your window controls.

When you’re finished, you should see your `CRecordset` variables listed next to the appropriate window control. Click OK.

11. Choose Build → Execute from the menu, or click Ctrl+F5. This builds and runs your program.

As you can see in Figure 5-9, it really is easy to develop a fully functional MFC database application using the MFC Application Wizard.

![Database application](image)

Figure 5-9: Using the 11 steps in this section, you are only minutes away from an MFC/ODBC database application.
The code for the version of ODBCDepartment can be found in the ODBCDepartmentA directory on the CD-ROM accompanying this book.

## Using Adds, Deletes, and Queries

The ODBCDepartment example in the previous section enabled scrolling through records as well as updates. In this section, you'll learn how to easily create add, delete, and query functionality inside an MFC ODBC application. Adding add, delete, and query functionality to an existing MFC/ODBC application such as the ODBCDepartment application requires the following six steps:

1. Pull up the menu from the Resources tab. Add the menu items in Table 5-1 to the Record menu item.

### Table 5-1 ODBCDepartment MENU ITEMS

<table>
<thead>
<tr>
<th>Text</th>
<th>ID</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;Add Record</td>
<td>ID_RECORD_ADDRECORD</td>
<td>Insert a new record.</td>
</tr>
<tr>
<td>&amp;Delete Record</td>
<td>ID_RECORD_DELETERECORD</td>
<td>Delete this record.</td>
</tr>
<tr>
<td>&amp;Query Record</td>
<td>ID_RECORD_QUERYRECORD</td>
<td>Find a department code.</td>
</tr>
</tbody>
</table>

Figure 5-10 shows how the menu should look when you're finished.

2. Edit your dialog box to include an edit box used for querying. The query box in Figure 5-11 includes an edit box named IDC_FINDCODE to contain the new department number that is used to query the database.
Figure 5-10: It’s good to add menu items when you add new functionality to a program.

Figure 5-11: ODBCDepartment needs an edit box for querying.
3. If you wish, make new toolbar buttons to tie to your new menu items. As you can see in Figure 5-12, three new toolbar items were added for adding, deleting, and querying.

You can open your Resources tab to find your toolbar. Not only can you paint new toolbar icons, but you can also cut and paste icons from existing toolbars into your toolbar.

After you’re done painting the toolbars, give them the same control name as your corresponding menu items. That way, you need to write only one set of code to handle both the menu and toolbar functions.

Figure 5-12: Toolbars can make running your program much easier.

4. Now use the Class Wizard to bind a new variable (m_FindDeptCode) to your new control (IDC_FINDCODE). Next, enter 4 in the Maximum Characters box (because department codes are limited to four characters). When you’re done, your member variables should look like those in Figure 5-13.
5. Click the Message Maps tab, and add an OnMove function to the CODBCDepartmentView class. Also add COMMAND functions for ID_RECORD_ADDRECORDER, ID_RECORD_DELETE_RECORD, and ID_RECORD_QUERY_RECORD. Also add an UPDATE_COMMAND_UI function for ID_RECORD_DELETE_RECORD so that you can disable delete functionality if you don’t have a record. Figure 5-14 shows how your Class Wizard should look after you complete this step.
6. At this point, we've come a long way without writing any Visual C++ code but rather using Visual C++ to generate code for us. However, to complete this program, you need to write five functions defined in step 5: OnRecordAddrecord, OnMove, OnRecordDeleterecord, OnUpdateRecordDeleterecord, and OnRecordQueryrecord.

**Writing OnRecordAddrecord and OnMove**

Writing the OnRecordAddrecord function is not difficult but requires the most knowledge of ODBC and the CRecordset class of any of the other functions. To add a record, you need to put the recordset into AddNew mode. Furthermore, you need a flag to indicate what mode you're currently in so that other functions know that you're currently trying to add a record. To incorporate add record functionality into your Visual C++ application, perform the following steps:

1. First, add the following BOOL flag to your CRecordView descendent class's header file. (In this case, it is the CODBCDepartmentView class in the ODBCDepartmentView.h header file):

   ```
   public:
   BOOL m_bAddingRecord;
   ```

2. Initialize the m_bAddingRecord in your CRecordView descendent constructor:

   ```
   CODBCDepartmentView::CODBCDepartmentView()
   : CRecordView(CODBCDepartmentView::IDD)
   {
   //{{AFX_DATA_INIT(CODBCDepartmentView)
   m_pSet = NULL;
   m_FindDeptCode = _T("");  // If not, set CRecordset in edit mode for updating
   }  //}}AFX_DATA_INIT
   m_bAddingRecord = FALSE;
   }
   ```

3. In the OnRecordAddrecord function in your CRecordView descendent class, you need to do the following:

   a. Update any program variables that currently need updating from the dialog box by using the UpdateData function, setting Edit mode if needed, and issuing an Update command:

      ```
      UpdateData(TRUE);  //Get data from dialog box
      if (!m_bAddingRecord) {
      //Currently adding?
      //If not, set CRecordset in edit mode for updating
      ```
b. Move to the last field using the `MoveLast` function. This ensures that you're not on the first record when you add:

```cpp
m_pSet->MoveLast(); //Get off record 1
```

c. Set your add flag to `TRUE`:

```cpp
m_bAddingRecord = TRUE; //Set flag
```

d. Clear the fields in the dialog window by setting them to `NULL`:

```cpp
m_pSet->SetFieldNull(NULL); //Clear all fields
```

e. Place the recordset in AddNew mode:

```cpp
m_pSet->AddNew(); //Set database in AddNew mode
```

f. Use the `UpdateData` function to display the cleared fields in the dialog box:

```cpp
UpdateData(FALSE); //Update dialog box fields
```

These steps can be seen in the following `OnRecordAddrecord` function:

```cpp
void CODBCDepartmentView::OnRecordAddrecord()
{
    CRecordsetStatus rStatus; //Status variable
    m_pSet->GetStatus(rStatus); //Get CRecordset status
    if (rStatus.m_lCurrentRecord >= 0) { //Records Exist?
        UpdateData(TRUE); //Get data from dialog box
        if (!m_bAddingRecord) { //Currently adding?
            //If not, set CRecordset in edit mode for updating
            m_pSet->Edit();
        }
        m_pSet->Update(); //Update data if needed
        m_pSet->MoveLast(); //Get off record 1
    }
    m_bAddingRecord = TRUE; //Set flag
    m_pSet->SetFieldNull(NULL); //Clear all fields
    m_pSet->AddNew(); //Set database in AddNew mode
}```
4. By default, the OnMove function updates the existing record before completing its move by placing the recordset in Edit mode and calling the Update function. However, because you are now enabling adds to the recordset, you need to override this action so you can add any new records that must be added. This can be done by the following additions to the OnMove function:

```cpp
BOOL CODBCDepartmentView::OnMove(UINT nIDMoveCommand)
{
    if (m_bAddingRecord) {    //Currently adding?
        //If so, Update the record before the move
        UpdateData(TRUE);    //Get data from dialog box
        m_pSet->Update();    //Update data if needed
        m_pSet->MoveLast();  //Go to the added record
        m_bAddingRecord = FALSE;    //Reset flag
    }
    //Continue with normal processing
    return CRecordView::OnMove(nIDMoveCommand);
}
```

Adding a new record is probably the most complicated database feature to understand. However, once you grasp how the MFC handles updates and how ODBC recordsets are placed in AddNew mode, the coding is not that difficult.

**Writing OnRecordDelete**

While adding a record takes the greatest understanding of MFC and ODBC, deleting a record requires the most handling and error checking. While the Delete function itself is relatively straightforward, there are other considerations whenever a record is deleted:

- You need to make sure that the user really wants to delete the record. You can do so with a message box asking if the user really wants the record deleted:

```cpp
if (AfxMessageBox(    //Be sure to verify your deletes
    "Are you sure you want to delete?",
    MB_YESNO) != IDYES) {
    return;
}
```
You have to make sure that the record really exists. If the user tries to delete a new record before adding it, then you merely cancel the add rather than delete a record:

```c++
if (m_bAddingRecord) { //Currently adding?
    //Don't delete, just cancel add.
    m_pSet->CancelUpdate();
    m_bAddingRecord = FALSE;
    m_pSet->MovePrev();
    return; }
```

You have to catch any errors that may occur during a delete. The CRecordset::Delete function throws a CDBException if there is a database error during the delete:

```c++
try {
    m_pSet->Delete(); //Delete record
} catch(CDBException* e1) { //Failed
    AfxMessageBox("Delete Failed:
    e1->m_strError,
    MB_ICONEXCLAMATION);
    m_pSet->MoveFirst(); //We lost our place.
    e1->Delete(); //Delete Error Message
    UpdateData(FALSE); //Update dialog box fields
    return;
}
```

---

**Error Trapping and try . . . catch Blocks**

C++ functions often return a code indicating an error, which is accomplished by a series of return codes that link several functions together. This is shown graphically in the following figure.

![Diagram of error trapping and try . . . catch blocks](image)

Traditional error processing required error chaining.
After a successful delete, you need to reposition the recordset so the user can still view records. If the last record is deleted, you should probably go to either the previous record or the first record. Otherwise, you should probably reposition the recordset to the next record:

```cpp
m_pSet->MoveNext();  //Go to next record
if (m_pSet->IsDeleted()) {
    //Was there a next record?
    m_pSet->MoveFirst();  //Deleted last record
}
```

The figure includes three functions: f1, f2, and f3. The f1 function calls the f2 function, which calls the f3 function. The f3 function returns an error that is captured by the f2 function and passed back to the f1 function. This technique is sometimes called error chaining.

Although error chaining is most often used in C++ programs for error handing, error chaining requires all linking functions to return the error code. If some function was not in a position to return an error, then valuable error information was lost.

try ... catch blocks (or just try blocks) are a later addition to the C++ standard. try ... catch blocks enable functions to “throw” errors that can be “caught” at any phase of the error chain. The following figure shows how try ... catch blocks change the error chain structure.

try ... catch blocks enable functions to “catch” any error that is “thrown.”

As you can see by the second figure, instead of returning a value through a series of functions, the later function simply throws an error. Any previous function in the error chain can catch this error and process it without the need for a return code.

You can catch as many errors as are thrown by placing multiple catch blocks after your try block. You can also throw an error within a try block, and it will be caught by the catch block immediately following the try block. try ... catch blocks simplify multiple error processing in complex situations.
Next, if you have deleted the last record, you probably need to set the fields up to add a new record because there are no records to display. This involves closing and opening the recordset to position the pointer off the deleted record and then calling the OnRecordAddrecord function written in the previous section to set up a new record:

```c++
try {
    if (m_pSet->IsDeleted()) { // Can't find a record
        AfxThrowDBException(SQL_ERROR,
                               m_pSet->m_pDatabase,
                               m_pSet->m_hstmt);
    }
}
// catch the error that was thrown
catch(CDBException* e2) { // No records exist
    AfxMessageBox("No more records", MB_ICONEXCLAMATION);
    e2->Delete(); // Delete Error Message
    // Close and Open to get rid of the Deleted record
    m_pSet->Close();
    m_pSet->Open();
    // No records, so set up an add record
    OnRecordAddrecord();
}
```

Finally, add an UpdateData(FALSE) statement to your function so that the dialog box is updated with the new values.

The code you need to add to the CODBCDepartmentView::OnRecordDelete record function is as follows:

```c++
void CODBCDepartmentView::OnRecordDeleteRecord() {
    if (AfxMessageBox( // Be sure to verify your deletes
                       "Are you sure you want to delete?",
                       MB_YESNO)
        != IDYES) {
        return;
    }
    if (m_bAddingRecord) { // Currently adding?
        // Don't delete, just cancel add.
        m_pSet->CancelUpdate();
        m_bAddingRecord = FALSE;
        m_pSet->MovePrev();
        return;
    }
    // Other code to handle deletion
}
```
Writing `OnUpdateRecordDeleterecord`

The `OnUpdateRecordDeleterecord` function is used to disable ("gray out") the Delete Record menu item and the corresponding toolbar item if a delete does not make sense. Here, you test using the `IsBOF`, `IsEOF`, and `IsDeleted` functions to see if there's a current record. If so, you enable the delete functionality. Adding an `Enable` function call to the `OnUpdateRecordDeleterecord` function can do this:
void CODBCDepartmentView::OnUpdateRecordDeleterecord(CCmdUI* pCmdUI)
{
    // Disable delete functionality if no record is found
    pCmdUI->Enable( // Enable delete if there's a record
        !m_pSet->IsBOF() &&
        !m_pSet->IsDeleted() &&
        !m_pSet->IsEOF());
}

Writing OnRecordQueryRecord

To add a query, you need to assign a value to the CRecordset::m_pSet->m__strFilter variable. The can be done with the following steps:

1. If you are in the middle of updating or adding a new record, you need to issue an Edit (unless you are adding a record and have already issued an AddNew):

    UpdateData(TRUE); // Get data from dialog box
    if (!m_bAddingRecord) { // Currently adding?
        // Set to update
        m_pSet->Edit();
    }
    m_pSet->Update(); // Update data if needed
    m_bAddingRecord = FALSE; // Reset flag

2. Check the bound filter variable (m_FindDeptCode) that was defined in step 4 at the beginning of this section, and if a value is in the field, form an SQL WHERE clause (without the WHERE) and assign it to a string variable:

    CString newFilter = ""; // Default is no filter
    if (m_FindDeptCode != "") {
        // Setup new filter
        newFilter = "DepartmentCode = '" + m_FindDeptCode + "'";
    }

3. Check to see if you need to apply the new filter. You need to apply the filter if it is different from the last filter used to perform the recordset query. The last filter is stored in the CRecordset::m_strFilter variable. If you reassign the filter, you must call the Requery function to apply the new filter:

    if (newFilter != m_pSet->m_strFilter) { // Filter has changed
        m_pSet->m_strFilter = newFilter; // Assign new filter
        if (!m_pSet->Requery()) { // Requery
            // Error occurred
            AfxMessageBox("Requery has failed");
    }
m_pSet->m_strFilter = ""; //Try to get back
m_pSet->Requery(); //Requery again
}
//Continue processing

4. Next, test to see if the new recordset has any records, and move to the first record, if possible. If not, display a message saying there are no records, and issue an Add by calling the OnRecordAddrecord function to clear the fields:

```cpp
try {
    //Go to the first record of the new filtered recordset
    m_pSet->MoveFirst();
}
catch(CDBException* e) {
    //Move failed because there are no records
    AfxMessageBox("No records were found", MB_ICONEXCLAMATION);
    e->Delete(); //Delete Error Message
    //No records, so set up an add record
    OnRecordAddrecord();
}
```

5. Finally, add an UpdateData(FALSE) statement to your function so that the dialog box is updated with the new values.

Put the code in the OnRecordQueryrecord function. The following code can be written to accomplish the tasks described in the preceding steps:

```cpp
void CODBDDepartmentView::OnRecordQueryrecord()
{
    CString newFilter = ""; //Default is no filter
    UpdateData(TRUE); //Get data from dialog box
    if (!m_bAddingRecord) { //Currently adding?
        //Set to update
        m_pSet->Edit();
    }
    m_pSet->Update(); //Update data if needed
    m_bAddingRecord = FALSE; //Reset flag
    if (m_FindDeptCode != "") {
        //Setup new filter
        newFilter = "DepartmentCode = '" + m_FindDeptCode + "'";
    }
    if (newFilter != m_pSet->m_strFilter) {
        //Filter has changed
```
m_pSet->m_strFilter = newFilter; //Assign new filter
if (!m_pSet->Requery()) {        //Requery
    //Error occurred
    AfxMessageBox("Requery has failed");
    m_pSet->m_strFilter = ";    //Try to get back
    m_pSet->Requery();           //Requery again
}
try {
    //Go to the first record of the new filtered recordset
    m_pSet->MoveFirst();
}
catch(CDBException* e) {   
    //Move failed because there are no records
    AfxMessageBox("No records were found",MB_ICONEXCLAMATION);
    e->Delete();       //Delete Error Message
    //No records, so set up an add record
    OnRecordAddrecord();
}

    UpdateData(FALSE);    //Update dialog box fields
}

When you’re finished, your application should look similar to the application shown in Figure 5-15.

![Image](image_url)

Figure 5-15: An example of an MFC ODBC program that supports adding, deleting, and querying

The code for the version of ODBCDepartment can be found in the ODBCDepartmentB directory on the CD-ROM that accompanies this book.
Saving and Transactions

Many modern commercial packages give the user the option to save or reject changes. This can be easily accomplished using database transactions. Transactions enable a developer to group together a series of database updates, and then either commit (make permanent) those changes to the database or roll back (erase) those changes from the database.

Access, Transactions, and CRecordset

Some databases act strangely with transactions. For instance, if you set up a transaction in Microsoft Access, you cannot have a cursor open. With traditional database programming techniques, it is not a problem, but with the MFC, it becomes a big problem.

With the MFC, you almost always have a CRecordset object open at all times. Behind this CRecordset object is a database cursor. As a result, you must close the CRecordset before opening a transaction:

```cpp
CRecordsetStatus rStatus; //Status variable
m_pSet->GetStatus(rStatus); //Get CRecordset status
m_pSet->Close(); //Close recordset and
cursor to start transaction
m_pSet->m_pDatabase->BeginTrans(); //Begin transaction
m_pSet->Open(); //Reopen the
recordset and cursor
//Restore record position
if (rStatus.m_lCurrentRecord >= 0) {
    m_pSet->SetAbsolutePosition(rStatus.m_lCurrentRecord+1);
}
```

Of course, the preceding technique doesn't work very well if you don't know the current record number. ODBC does not count records and keeps track of records only if you scroll through them from beginning to end. However, you could code an application-specific routine that searched for a given key.

If your transaction processing does not appear to be working, be sure to check with your DBA or your database documentation to examine cursor behavior.
When the user indicates that the changes are to be saved, a `CommitTrans` function is called. When the user decides to discard the changes, a `Rollback` function is called. This can be accomplished using the following steps:

1. You need a Save menu function. You can add this function yourself, or you can generate a new project using the techniques discussed earlier in this chapter and choose the Database view with file support option in Step 2 of 6 in the MFC App Wizard (Figure 5-16). Then generate this project with the same options you used at the beginning of this chapter.

![Figure 5-16: The Database view with a file support option adds file functionality to your MFC window.](image)

2. If you choose to generate a new project, you may need to delete some generated File options (such as Open) that make little sense with your application. You may also want to use the New menu option to add a record rather than the Add Record menu option used earlier in this chapter. In Figure 5-17, you can see that the File menu option has only three choices: New, Save, and Exit.

   Because the New menu choice is now being used for new records, the Add Record menu choice is no longer needed. As you can see in Figure 5-18, only the Delete Record and Query Record choices remain.

3. Edit your toolbar to include any additional functionality (Figure 5-19).

4. Use the Class Wizard to add the following functionality to your `CRecordView` descendent (`C0DBCDepartmentView`). You should use the Add Function button to create the functions listed in Table 5-2.
Figure 5-17: Not all the file functionality is needed for every project.

Figure 5-18: File functionality should not be duplicated; therefore, the Add Record choice is no longer needed.
Figure 5-19: Toolbars need to be adjusted to show additional functionality.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Object ID</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnRecordDeleteRecord</td>
<td>ID_RECORD_DELETEREcord</td>
<td>COMMAND</td>
</tr>
<tr>
<td>OnUpdateRecordDelete record</td>
<td>ID_RECORD_DELETEREcord</td>
<td>UPDATE_COMMAND_UI</td>
</tr>
<tr>
<td>OnRecordQueryRecord</td>
<td>ID_RECORD_QUERYRECORD</td>
<td>COMMAND</td>
</tr>
<tr>
<td>OnMove</td>
<td>CODBCDepartmentView</td>
<td>COMMAND</td>
</tr>
<tr>
<td>OnFileNew</td>
<td>ID_FILE_NEW</td>
<td>COMMAND</td>
</tr>
<tr>
<td>OnFileSave</td>
<td>ID_FILE_SAVE</td>
<td>COMMAND</td>
</tr>
<tr>
<td>OnUpdateFileSave</td>
<td>ID_FILE_SAVE</td>
<td>UPDATE_COMMAND_UI</td>
</tr>
<tr>
<td>OnDestroy</td>
<td>CODBCDepartmentView</td>
<td>ON_WM_DESTROY</td>
</tr>
<tr>
<td>OnChangeDepartment code</td>
<td>IDC_DEPARTMENTCODE</td>
<td>EN_CHANGE</td>
</tr>
<tr>
<td>OnChangeDepartment name</td>
<td>IDC_DEPARTMENTNAME</td>
<td>EN_CHANGE</td>
</tr>
</tbody>
</table>
5. Code the events you described in step 4. The **OnMove**, **OnRecordDelete** record, **OnUpdateRecordDelete** record, and **OnRecordQuery** record functions are identical to the functions with the same names listed in Table 5-2, with one exception: instead of calling an **OnRecordAdd** record as before, now you call the **OnFileNew** function. Code for the **OnFileNew** function is identical to the code for the previously described **OnAddRecord** function.

6. You need to flag when you are in the middle of an add, as before. You also need to flag when a change is made. This can be done by defining public class variables in the **CODBCDepartmentView** class:

```cpp
public:
    BOOL m_bAddingRecord;
    BOOL m_bFieldsChanged;
```

7. **Initialize your Boolean variables to** **FALSE in the** **CODBCDepartmentView** **constructor**:

```cpp
CODBCDepartmentView::CODBCDepartmentView()
    : CRecordView(CODBCDepartmentView::IDD)
{
    //AFX_DATA_INIT(CODBCDepartmentView)
    m_pSet = NULL;
    m_FindDeptCode = _T("*");
    //}AFX_DATA_INIT
    m_bAddingRecord = FALSE;
    m_bFieldsChanged = FALSE;
}
```

8. **Next, add code to the** **OnChangeDepartmentcode** **and** **OnChangeDepartmentname** **functions to set the** **m_bFieldsChanged** **to** **TRUE in the** **CODBCDepartmentView** **class**:

```cpp
void CODBCDepartmentView::OnChangeDepartmentcode()
{
    m_bFieldsChanged = TRUE;
}
void CODBCDepartmentView::OnChangeDepartmentname()
{
    m_bFieldsChanged = TRUE;
}
```

9. **Use the** **OnUpdateFileSave** **function to disable the Save button and Save menu choice if no changes have been made.**

```cpp
void CODBCDepartmentView::OnUpdateFileSave(CCmdUI* pCmdUI)
{
    //Enable save functionality if a record has changed
```
pCmdUI->Enable(m_bFieldsChanged);
}

10. Open your transaction before processing in the `OnInitialUpdate` function inside the `CODBCDepartmentView` class:

```cpp
void CODBCDepartmentView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_oDBCDepartmentSet;
    CRecordView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();
    m_pSet->Close(); // Close to start transaction
    if (m_pSet->m_pDatabase->CanTransact()) {
        m_pSet->m_pDatabase->BeginTrans();
    }
    m_pSet->Open(); // Open after starting transaction
}
```

11. Write the `OnFileSave` function. This consists of the following steps:

   a. Make sure transactions are enabled and that fields have changed:
      ```cpp
      if (m_bFieldsChanged && m_pSet->CanTransact()) {
      
      }
      ```

   b. Issue an update to update the current record, if needed. Make sure the recordset is in either Edit mode or AddNew mode first:
      ```cpp
      if (!m_bAddingRecord) { // Currently adding?
        m_pSet->Edit(); // If not, set edit mode
      }
      UpdateData(TRUE); // Get data from dialog box
      m_bAddingRecord = FALSE; // Reset flag
      m_pSet->Update(); // Update data if needed
      ```

   c. Commit the transaction to permanently save any changes made:
      ```cpp
      m_pSet->m_pDatabase->CommitTrans(); // Commit changes
      ```

   d. After you issue a CommitTrans or a Rollback function, the transaction ends. You need to restart the transaction again by using a BeginTrans:
      ```cpp
      m_pSet->m_pDatabase->BeginTrans(); // Start Transaction
      m_bFieldsChanged = FALSE; // Reset flag
      ```

   e. The code to do this for a Microsoft Access database (which includes the closing of the `CRecordset` before beginning a transaction) is as follows:
      ```cpp
      void CODBCDepartmentView::OnFileSave()
      {
      ```
if (m_bFieldsChanged && m_pSet->CanTransact()) {
    if (!m_bAddingRecord) {    //Currently adding?
        m_pSet->Edit();    //If not, set edit mode
    }
    UpdateData(TRUE);    //Get data from dialog box
    m_bAddingRecord = FALSE;    //Reset flag
    m_pSet->Update();    //Update data if needed

    //Commit changes
    m_pSet->m_pDatabase->CommitTrans();
    CRecordsetStatus rStatus;    //Status variable
    m_pSet->GetStatus(rStatus);    //Get CRecordset status
    m_pSet->Close();    //Close to start transaction

    //Start Transaction
    m_pSet->m_pDatabase->BeginTrans();
    m_pSet->Open();            //Reopen CRecordset
    //Restore record position
    if (rStatus.m_lCurrentRecord >= 0) {
        m_pSet->SetAbsolutePosition(rStatus.m_lCurrentRecord+1);
    }
    m_bFieldsChanged = FALSE;    //Reset flag
    UpdateData(FALSE);    //Update dialog box fields
}
}

12. With certain databases, uncommitted changes are erased when the connection terminates. Other databases default to automatically committing those changes when the transaction terminates. To ensure that neither behavior occurs, when a user terminates your program, you need to check to see if there are uncommitted changes, which can be done in the OnDestroy function using the following steps:

   a. Check if transactions are supported and if the fields have changed:

   if (m_pSet->CanTransact()) {
       if (m_bFieldsChanged) {
           //Handle changes
       } else {
           //No changes, but close the transaction with a commit.
           m_pSet->m_pDatabase->CommitTrans();
       }
   }
b. Where the //Handle changes comment is in the preceding code, you know that changes have been made. You must check to see if the user wants these changes saved and, if not, roll back your changes:

```cpp
if (AfxMessageBox (
    "Records have been changed. Do you want to save?",
    MB_YESNO) == IDNO) {
    m_pSet->m_pDatabase->Rollback();
}
```

c. If the user clicks Yes in response to the message box in the preceding code, you must accept any current changes. As with the OnFileSave function, this entails making sure the recordset is in AddNew or Edit mode, accepting any changes, updating the database, and committing the transaction:

```cpp
if (!m_bAddingRecord) {    //Currently adding?
    m_pSet->Edit();    //If not, set edit mode
}
UpdateData(TRUE);      //Get data from dialog box
m_pSet->Update();      //Update data if needed
// Now Commit the changes to the database
m_pSet->m_pDatabase->CommitTrans();
```

These changes should be added to the OnDestroy function in the CODBC DepartmentView class:

```cpp
void CODBCDepartmentView::OnDestroy()
{

//Check for changed fields and transaction ability
if (m_pSet->CanTransact()) {
    if (m_bFieldsChanged) {
        if (AfxMessageBox (  
            "Records have been changed. Do you want to save?",
            MB_YESNO) == IDNO) {
            m_pSet->m_pDatabase->Rollback();
        }
    }
    else {
        if (!m_bAddingRecord) {    //Currently adding?
            //If not, set edit mode
            m_pSet->Edit();
        }
        UpdateData(TRUE);      //Get data from dialog box
        m_pSet->Update();      //Update data if needed
        m_pSet->m_pDatabase->CommitTrans();
    }
}
```
else {
    // No changes, but close the transaction with a commit.
    m_pSet->m_pDatabase->CommitTrans();
}
CRecordView::OnDestroy();

When you’re finished, your program should look similar to the one shown in Figure 5-20.

![Figure 5-20: The ODBCDepartment application now uses transactions to support saving the data.](image)

When you leave the ODBCDepartment application after making changes, the program prompts you with a message box such as the one shown in Figure 5-21 to see if you want to save your changes.

![Figure 5-21: It’s good to prompt users to see if they want to save their changes.](image)

Certain databases, such as Microsoft SQL Server, limit the size of their transaction log. A transaction log often fills up if too many database updates are done without committing or rolling back. Be sure to check your database documentation to see your transaction limits, and don’t enable numerous database updates over an extended period of time without committing those changes.

The code for the version of ODBCDepartment can be found in the ODBCDepartmentC directory on the CD-ROM.

Using Joins

You often need to bring in more than one table for display. For instance, say you wanted to see and update instructor records as you scrolled through each department. The simplest way to achieve this is through a table join.

Development using table joins with the MFC Application Wizard is almost identical to development with a single table. Perform the following steps:

1. Use the MFC Application Wizard. In the Step 2 of 6 dialog box, choose Database view without file support. Then click the Data Source button.

2. In the Database Options dialog box, choose ODBC and Dynaset. Then choose your ODBC data source. When the Select Database Tables dialog box opens, choose two or more tables to be joined together. In Figure 5-22, the Department table and the Instructor table are chosen for the join.

3. In the Step 4 of 6 dialog box, deselect (turn off) printing and print preview. Accept the rest of the defaults.

4. Paint your screen. Don’t forget to rename your edit boxes to something more readable. Figure 5-23 shows the dialog box designed for this table inquiry.
Figure 5-23: In the dialog box, add all the fields needed for all the tables in your join.

5. Use the Class Wizard to bind database fields to your edit boxes, as shown in Figure 5-24.

Figure 5-24: All the columns should be available for selecting.
Notice that, when you have duplicate column names, such as DepartmentCode, that you have two variables that correspond to these columns (m_pSet->DepartmentCode and m_pSet->DepartmentCode2). If you are joining on the field with the same name, then simply picking one of the columns to bind is fine because both columns, in a join, should contain the same value. However, if you use columns that have the same name ("Name","Type", and so on) that are not part of a join, you have to be careful which field is bound to which table. The DoFieldExchange function that is overridden in your CRecordset descendant should show you fully qualified column names (when needed) and the variable where the column name is bound.

6. Next, add the join criteria using the m_strFilter string in the CRecordset descendant constructor. In the following code, you can see that the CODBCDeptInstrSet constructor contains a line that tells ODBC how to join the two related tables.

CODBCDeptInstrSet::CODBCDeptInstrSet(CDatabase* pdb)
: CRecordset(pdb)
{
    //{{AFX_FIELD_INIT(CODBCDeptInstrSet)
    m_DepartmentCode = _T(""");
    m_DepartmentName = _T(""");
    m_InstructorID = 0;
    m_Name = _T(""");
    m_DepartmentCode2 = _T(""");
    m_EMAIL = _T(""");
    m_Notes = _T(""");
    m_nFields = 7;
    //}}AFX_FIELD_INIT
    m_nDefaultType = dynaset;
    m_strFilter = "Instructor.DepartmentCode = Department.DepartmentCode";
}

When you're done, compile your program. As Figure 5-25 shows, when you scroll through your record, you can see values from both tables.
Using Multiple ODBC Recordsets

It's much simpler to use a single recordset for every dialog box than to use multiple recordsets in a dialog box. In the last example, you could create a joined recordset in mere minutes. However, there are some problems with this approach:

- You cannot, by default, update your recordset. With a joined recordset in ODBC, the columns are not updateable. This also makes adding or deleting records somewhat problematic because the recordset won't know which table to add to or delete from.

- Depending on your join, your results may contain duplicate values (for example, many of the same department codes and names for each instructor).

To get around this restriction, you may want to open more than one related CRecordset classes at the same time: each containing only one table. In Figure 5-26, you can see that there are two CRecordset classes open at once; one CRecordset for the Department table, and then a related CRecordset that opens all the related instructors for that department.
Figure 5-26: You often need to open related CRecordset classes at the same time.

The layout for Figure 5-26 is shown in Figure 5-27. Figure 5-27 contains one independent CRecordset object that uses the Department table. Further below are all the related instructors (if any) for that department. Whenever the department record changes, the related CRecordset must be immediately requeried to be populated with corresponding Instructor records.

Figure 5-27: The formats of a one-to-many display, also called a “drill down” format, are common in many custom and commercial applications.

The following sections show how to code for multiple related CRecordset objects. In this part of the chapter, we take the ODBCDepartment project and add code for a second Instructor table recordset.
Creating a Second CRecordset Class

When using multiple CRecordset classes, the first task to complete is to create a second CRecordset object. This can be done using the Class Wizard and the following steps:

1. Click View → ClassWizard, and click the Add Class button. Choose New from the pop-up menu.

2. In the New Class dialog box, enter the name of your new class (CODBCInstructorSet), then choose CRecordset as the Base class, and click OK, as shown in Figure 5-28.

![Figure 5-28: The New Class dialog box enables you to create a new class from an existing base class and add it to your project.](image)

3. Because you chose a base class of CRecordset, the Database Options dialog box then opens, as shown in Figure 5-29. Choose ODBC, and pick the ODBC database you want to use. This database should be the same as the database you picked before for your primary CRecordset class.

4. Next, you should pick your table (Instructor) from your database using the Select Database Tables dialog box shown in Figure 5-30. This should be a table that relates to the primary table somehow. After you choose this table, you should return to the Class Wizard main window. Click OK to return to your Visual C++ environment. Your new class is now added to your source files.
5. The only change you need to make to your second CRecordset class is
to add a parameter. This is a pointer that will, eventually, point to the
related field in the other CRecordset object. First, declare your pointer
in the header file of your new class (CODBCInstructor.h):

```cpp
//Parameters
public:
    CString *m_pDepartmentCode;
```

Next, indicate the number of parameters in the constructor of your new
class (CODBCInstructor.cpp) by assigning the number of related columns
(usually 1) to the m_nParams class variable:

```cpp
CODBCInstructorSet::CODBCInstructorSet(CDatabase* pdb)
    : CRecordset(pdb)
{
    //AFX_FIELD_INIT(CODBCInstructorSet)
    m_InstructorID = 0;
    m_Name = _T(""):
    m_DepartmentCode = _T(""):
    m_EMAIL = _T(""):
    m_Notes = _T(""):
```
m_nFields = 5;
//}}AFX_FIELD_INIT
m_nParams = 1;
m_nDefaultType = dynaset;

Finally, assign the parameter in the DoFieldExchange class (CODBCInstructor.cpp). Assign a WHERE clause string to the m_strFilter function to indicate the field exchange that you are going to use to bind a parameter, and then use the appropriate RFX function to bind your pointer as a parameter.

```cpp
void CODBCInstructorSet::DoFieldExchange(CFieldExchange* pFX) {
    //((AFX_FIELD_MAP(CODBCInstructorSet)
pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[InstructorID]")), m_InstructorID);
    RFX_Text(pFX, _T("[Name]")), m_Name);
    RFX_Text(pFX, _T("[DepartmentCode]")), m_DepartmentCode);
    RFX_Text(pFX, _T("[EMAIL]")), m_EMAIL);    
    RFX_Text(pFX, _T("[Notes]")), m_Notes);
    //})AFX_FIELD_MAP
    m_strFilter = "DepartmentCode = ?";    //Assign a filter
    //Indicate that you're binding parameters
    pFX->SetFieldType(CFieldExchange::param);
    //Now bind the parameter to the m_pDepartmentCode pointer
    //Filter parameter names (i.e. "Parm1") aren't important
    RFX_Text(pFX, _T("Parm1"), *m_pDepartmentCode);
}
```

### Implementing Changes to CRecordView Class

Your record view requires the most changes when you add a second recordset to your record view. Not only do you need to make changes to the dialog, menu, and toolbar resources, but you also need to instantiate and deallocate your new recordset and program all the additional functionality yourself.

**IMPLEMENTING CHANGES TO RESOURCES WITH THE CLASS WIZARD**

Although resources are not technically part of the CRecordView class, the CRecordView class interacts with resources more than any other MFC class in your application. When adding any fields to a database, you need to change your dialog boxes and menu items to accommodate the new changes.
The Instructor table has five columns: InstructorID, Name, EMAIL, Notes, and DepartmentCode. Because the Instructor table is being joined to the Department table through the DepartmentCode column, there is no need to redisplay it. Also, because InstructorID is an autoincrement field used to create a primary key, there is no need to show that either. The other fields (Name, EMAIL, and Notes) should be added to your dialog box, as shown in Figure 5-31.

Figure 5-31: New fields from the related recordset must be added to your system.

Next, you need to add menu functionality. Earlier, you used the default Records menu choice because you had only one set of records. Now, with both Department records and Instructor records, you probably want to differentiate. However, rename only the menu options, and don't rename their control ID, because it is used to call functions that are already written. Figure 5-32 shows how Record was renamed to Department and an Instructor menu option was added with all the appropriate adding, deleting, and scrolling functionality.

Finally, your toolbar should also have some functionality dedicated to the new recordset. As you can see in Figure 5-33, two sets of database options (one color-coded green, and one color-coded purple when you run the executable) are added to the toolbar. This way, one can be used for Department recordset functionality while the other can be used for Instructor recordset functionality.
Figure 5-32: Your new recordset needs its own functionality.

Figure 5-33: Toolbars can be tied to the new menu options for toolbar support of your new recordset.
Table 5-3 shows the functions you'll need to define for the new menu structure, and which ones are new to this multi-record example. You should use the class wizard to generate these commands for you so you don't need to mess with the definitions.

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>New</th>
<th>Function</th>
<th>Object ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New</td>
<td>No</td>
<td>OnFileNew</td>
<td>ID_FILE_NEW</td>
</tr>
<tr>
<td>Save</td>
<td>No</td>
<td>OnFileSave</td>
<td>ID_FILE_SAVE</td>
</tr>
<tr>
<td>Save</td>
<td>No</td>
<td>OnUpdateFileSave</td>
<td>ID_FILE_SAVE</td>
</tr>
<tr>
<td>Department</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete Record</td>
<td>No</td>
<td>OnRecordDeleteRecord</td>
<td>ID_RECORD_DELETERECORD</td>
</tr>
<tr>
<td>Delete Record</td>
<td>No</td>
<td>OnUpdateRecordDeleteRecord</td>
<td>ID_RECORD_DELETERECORD</td>
</tr>
<tr>
<td>Query Record</td>
<td>No</td>
<td>OnRecordQueryRecord</td>
<td>ID_RECORD_QUERYRECORD</td>
</tr>
<tr>
<td>Instructor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First Instructor</td>
<td>Yes</td>
<td>OnInstructorFirstRecord</td>
<td>ID_INSTRUCTOR_FIRSTRECORD</td>
</tr>
<tr>
<td>First Instructor</td>
<td>Yes</td>
<td>OnUpdateInstructorFirstRecord</td>
<td>ID_INSTRUCTOR_FIRSTRECORD</td>
</tr>
<tr>
<td>Previous Instructor</td>
<td>Yes</td>
<td>OnInstructorPreviousRecord</td>
<td>ID_INSTRUCTOR_PREVIOUSRECORD</td>
</tr>
<tr>
<td>Previous Instructor</td>
<td>Yes</td>
<td>OnUpdateInstructorPreviousRecord</td>
<td>ID_INSTRUCTOR_PREVIOUSRECORD</td>
</tr>
<tr>
<td>Next Instructor</td>
<td>Yes</td>
<td>OnInstructorNextRecord</td>
<td>ID_INSTRUCTOR_NEXRECORD</td>
</tr>
<tr>
<td>Next Instructor</td>
<td>Yes</td>
<td>OnUpdateInstructor</td>
<td>ID_INSTRUCTOR_NEXRECORD</td>
</tr>
</tbody>
</table>
Chapter 5: Developing ODBC Projects

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>New</th>
<th>Function</th>
<th>Object ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Instructor</td>
<td>Yes</td>
<td>OnInstructorLastRecord</td>
<td>ID_INSTRUCTOR_LASTRECORD</td>
</tr>
<tr>
<td>Last Instructor</td>
<td>Yes</td>
<td>OnUpdateInstructorLastRecord</td>
<td>ID_INSTRUCTOR_LASTRECORD</td>
</tr>
<tr>
<td>Add Instructor</td>
<td>Yes</td>
<td>OnInstructorAddInstructor</td>
<td>ID_INSTRUCTOR_ADDINSTRUCTOR</td>
</tr>
<tr>
<td>Delete Instructor</td>
<td>Yes</td>
<td>OnInstructorDeleteInstructor</td>
<td>ID_INSTRUCTOR_DELETEINSTRUCTOR</td>
</tr>
<tr>
<td>Delete Instructor</td>
<td>Yes</td>
<td>OnUpdateInstructorDeleteInstructor</td>
<td>ID_INSTRUCTOR_DELETEINSTRUCTOR</td>
</tr>
</tbody>
</table>

In addition to the menu options defined by the Class Wizard, the control event handlers in Table 5-4 should also be created in order to trap for transactions.

**Table 5-4 Control Events Used for Transactions**

<table>
<thead>
<tr>
<th>New</th>
<th>Function</th>
<th>Object ID</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>OnDestroy</td>
<td>CODBCDepartmentView</td>
<td>WM_DESTROY</td>
</tr>
<tr>
<td>No</td>
<td>OnChangeDepartmentcode</td>
<td>IDC_DEPARTMENTCODE</td>
<td>EN_CHANGE</td>
</tr>
<tr>
<td>No</td>
<td>OnChangeDepartmentname</td>
<td>IDC_DEPARTMENTNAME</td>
<td>EN_CHANGE</td>
</tr>
<tr>
<td>Yes</td>
<td>OnChangeEmail</td>
<td>IDC_EMAIL</td>
<td>EN_CHANGE</td>
</tr>
<tr>
<td>Yes</td>
<td>OnChangeInstructorname</td>
<td>IDC_INSTRUCTORNAME</td>
<td>EN_CHANGE</td>
</tr>
<tr>
<td>Yes</td>
<td>OnChangeNotes</td>
<td>IDC_NOTES</td>
<td>EN_CHANGE</td>
</tr>
</tbody>
</table>

**NEW FUNCTIONS NEEDED FOR THE CRECORDVIEW.H HEADER FILE**

After you use the Class Wizard to add functions to your project, your CrecordView.h header file (CODBCDepartmentView.h) should have many functions added to it. However, you still need to add the following:
You need to add an include statement to the header file you just created:

```cpp
#include "ODBCInstructorSet.h"
```

You need to add a Boolean variable to indicate if you are currently adding a record to your new recordset:

```cpp
public:
    BOOL m_bAddingInstructor;
```

You need to add function prototypes to the following functions:

```cpp
public
    void OpenInstructorRecordset(long nRows = 0);
    void UpdateInstructor();
    void AddInstructor();
```

You need to create a pointer to the new recordset. This should be done after the `//{{AFX_DATA(CODBCDepartmentView)}` comment so that the Class Wizard still functions after you bind your variables.

```cpp
public:
    ////{{AFX_DATA(CODBCDepartmentView)
    enum { IDD = IDD_ODBCDEPARTMENT_FORM };  
    CODBCDepartmentSet* m_pSet;  
    CString m_FindDeptCode;  
    ////}}AFX_DATA
    CODBCInstructorSet* m_pInstructorSet;
```

NEW FUNCTIONS NEEDED FOR THE CRecordView

Several functions have to be written or modified to handle the second recordset. In addition, you must code those functions that you generated from the Class Wizard, which can be done using the following steps:

1. Just as with the primary recordset, you need a function to place the second recordset into AddNew mode and to set up the display to add a new record. The following AddInstructor function sets the add flag, clears out the instructor fields, and places the InstructorSet CRecordset in AddNew mode:

```cpp
void CODBCDepartmentView::AddInstructor()
{
    m_bAddingInstructor = TRUE;  //Set flag
    m_pInstructorSet->SetFieldNull(NULL);  //Clear fields
    //Set database in AddNew mode
    m_pInstructorSet->AddNew();
    UpdateData(FALSE);  //Update dialog box fields
}
```
The **AddInstructor** function is called in three situations:

- The user requests to add an instructor through the menu or a toolbar button.
- The user requests to add a new department.
- The user scrolls through a department, but there are no related instructor records. In this case, the dialog box is set up to add an instructor. If no changes are made, the **CRecordset::Update** function won't write the new record to the database.

2. You need a function to open your recordset both initially and after each transaction **BeginTrans** call. The **OpenInstructorRecordset** is used to initially open the InstructorSet recordset and to reopen the recordset after a transaction begins:

```cpp
void CODBCDepartmentView::OpenInstructorRecordset(long nRows)
{
    m_pInstructorSet->Open();
    m_pInstructorSet->Requery();
    CRecordsetStatus rStatus; //Status variable
    //Get CRecordset status
    m_pInstructorSet->GetStatus(rStatus);
    //no records Exist?
    if (rStatus.m_lCurrentRecord < 0) {
        //No instructor records, so add one.
        AddInstructor();
    }
    else {
        this -> m_pInstructorSet ->
        SetAbsolutePosition(nRows+1);
    }
}
```

3. You need a function to handle your updates. The following **UpdateInstructor** function shows how an update should be written for your second recordset:

```cpp
void CODBCDepartmentView::UpdateInstructor()
{
    if (!m_bAddingInstructor) {
        if (m_pInstructorSet->IsBOF() ||
            m_pInstructorSet->IsEOF() ||
            m_pInstructorSet->IsDeleted()) {
            return;
        }
    }
```
4. With the Class Wizard, you should have indicated that you wanted to trap certain OnChange functions. That way, you can set a flag to indicate that changes have been made and that you need to ask if the user wants to save their changes. In the following three functions, the \texttt{m\_bFieldsChanged} flag is set when changes are made in any of the edit boxes:

```cpp
def void CODBCDepartmentView::OnChangeEmail()
{
    m_bFieldsChanged = TRUE;
}
def void CODBCDepartmentView::OnChangeInstructorname()
{
    m_bFieldsChanged = TRUE;
}
def void CODBCDepartmentView::OnChangeNotes()
{
    m_bFieldsChanged = TRUE;
}
```

5. Add code for the Add Instructor menu choice. When the user wishes to add an instructor, first call a function to update the instructor in case some changes were made to the existing window. Then call a function to add an instructor:

```cpp
def void CODBCDepartmentView::OnInstructorAddinstructor()
{
    UpdateInstructor();
    AddInstructor();
}
```

6. Add code for the Delete Instructor menu choice using the same technique you used when coding the Delete Department menu choice:

```cpp
def void CODBCDepartmentView::OnInstructorDeleteinstructor()
{
    if (AfxMessageBox( //Be sure to verify your deletes
        "Are you sure you want to delete this instructor?",
        MB_YESNO)
        != IDYES) {
```
return;
}
if (m_bAddingInstructor) ( //Currently adding?
  //Don't delete, just cancel add.
  m_pInstructorSet->CancelUpdate();
  m_bAddingInstructor = FALSE;
  OnInstructorPreviousRecord();
  return;
}
try {
  m_pInstructorSet->Delete(); //Delete record
} catch(CDBException* e1) ( //Failed
  AfxMessageBox("Delete Failed:
  e1->m_strError,
  MB_ICONEXCLAMATION);
  e1->Delete(); //Delete Error Message
  OnInstructorFirstRecord(); //Recapture place
  return;
}
try {
  OnInstructorNextRecord(); //Go to the next record
  if (m_pSet->IsDeleted()) { //Next record exists?
    //Deleted last record
    OnInstructorFirstRecord();
  }
  if (m_pSet->IsDeleted()) { //Can't find a record
    AfxThrowDBException(SQL_ERROR,
      m_pInstructorSet->m_pDatabase,
      m_pInstructorSet->m_hstmt);
  }
  UpdateData(FALSE); //Update dialog box fields
} catch(CDBException* e2) ( //No records exist
  AfxMessageBox(
    "No more Instructors for this department",
    MB_ICONEXCLAMATION);
  e2->Delete(); //Delete Error Message
  //Close and Open to get rid of the Deleted record
  m_pInstructorSet->Close();
  m_pInstructorSet->Open();
  //No records, so set up an add record
  OnInstructorAddInstructor();
}
7. You need to code for the CRecordset record movement menu choices (that is, First Instructor, Previous Instructor, Next Instructor, and Last Instructor). This entails first updating the instructor, then moving the CRecordset to the appropriate position in the recordset, and then updating the dialog box from CRecordset. The following four functions do this:

```c
void CODBCDepartmentView::OnInstructorFirstrecord()
{
    UpdateInstructor();
    m_pInstructorSet->MoveFirst();
    UpdateData(FALSE); //Update dialog from database
}

void CODBCDepartmentView::OnInstructorLastrecord()
{
    UpdateInstructor();
    m_pInstructorSet->MoveLast();
    UpdateData(FALSE); //Update dialog from database
}

void CODBCDepartmentView::OnInstructorNextrecord()
{
    UpdateInstructor();
    m_pInstructorSet->MoveNext();
    if (m_pInstructorSet->IsEOF()) {
        m_pInstructorSet->MoveLast();
    }
    UpdateData(FALSE); //Update dialog from database
}

void CODBCDepartmentView::OnInstructorPreviousrecord()
{
    UpdateInstructor();
    m_pInstructorSet->MovePrev();
    if (m_pInstructorSet->IsBOF()) {
        m_pInstructorSet->MoveFirst();
    }
    UpdateData(FALSE); //Update dialog from database
}
```

8. You need to gray out menu and toolbar choices when they do not apply:

- You cannot delete a record when you are not positioned on a record or there is no current record:

```c
void CODBCDepartmentView::OnUpdateInstructorDeleteinstructor(
    CCmdUI* pCmdUI)
{
```
//Disable delete functionality if no record is found
pCmdUI->Enable( //Enable delete if there's a record
   !m_pInstructorSet->IsBOF() &&
   !m_pInstructorSet->IsDeleted() &&
   !m_pInstructorSet->IsEOF());

You cannot go to the first record or the previous record if you are already at the first record:

void CODBCDepartmentView::OnUpdateInstructorFirstrecord(CCmdUI* pCmdUI)
{
  CRecordsetStatus rStatus; //Status variable
  //Get CRecordset status
  m_pInstructorSet->GetStatus(rStatus);
  pCmdUI->Enable( //Disable at first record
      rStatus.m_lCurrentRecord > 0);
}

void CODBCDepartmentView::OnUpdateInstructorPreviousrecord(CCmdUI* pCmdUI)
{
  CRecordsetStatus rStatus; //Status variable
  m_pInstructorSet->GetStatus(rStatus);
  //Get CRecordset status
  pCmdUI->Enable( //Disable at first record
      rStatus.m_lCurrentRecord > 0);
}

* You cannot go to the last record or the next record if you are already at the last record:

void CODBCDepartmentView::OnUpdateInstructorLastrecord(CCmdUI* pCmdUI)
{
  CRecordsetStatus rStatus; //Status variable
  m_pInstructorSet->GetStatus(rStatus);
  //Get CRecordset status
  if (rStatus.m_bRecordCountFinal) {
    long recs = m_pInstructorSet->GetRecordCount() - 1;
    pCmdUI->Enable( //Disable at last record
      recs > 0);
  }

CHANGES NEEDED TO EXISTING CRecordView FUNCTIONS
Finally, you must update existing functions that you’ve already used for your single recordset. Do this with the following steps:

1. Update the CRecordView constructor to set your new recordset to NULL and to initialize your Boolean adding variable:

   CODBCDepartmentView::CODBCDepartmentView()
   : CRecordView(CODBCDepartmentView::IDD)
   {
     //AFX_DATA_INIT(CODBCDepartmentView)
     m_pSet = NULL;
     m_FindDeptCode = _T("");
     //AFX_DATA_INIT
     m_bAddingRecord = FALSE;
     //Next two lines added by Chuck Wood for
     //instructor table support
     m_pInstructorSet = NULL;
     m_bAddingInstructor = FALSE;
   
   }
2. Edit the DoDataExchange function to bind records to the new recordset. Note that, while you can use the Class Wizard to do this, the new recordset variables won’t appear on your drop-down list of choices. It may be easier to type them in yourself:

```cpp
void CODBCDepartmentView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    //AFX_DATA_MAP(CODBCDepartmentView)
    DDX_Text(pDX, IDC_FINDCODE, m_FindDeptCode);
    DDV_MaxChars(pDX, m_FindDeptCode, 4);
    DDX_FieldText(pDX, IDC_DEPARTMENTCODE, m_pSet->m_DepartmentCode, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_DepartmentCode, 4);
    DDX_FieldText(pDX, IDC_DEPARTMENTNAME, m_pSet->m_DepartmentName, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_DepartmentName, 50);
    //}AFX_DATA_MAP
    DDX_Text(pDX, IDC_EMAIL, m_pInstructorSet->m_EMAIL);
    DDV_MaxChars(pDX, m_pInstructorSet->m_NAME, 50);
    DDX_Text(pDX, IDC_INSTRUCTORNAME, m_pInstructorSet->m_Name);
    DDV_MaxChars(pDX, m_pInstructorSet->m_Name, 50);
    DDX_Text(pDX, IDC_NOTES, m_pInstructorSet->m_Notes);
    DDV_MaxChars(pDX, m_pInstructorSet->m_Notes, 50);
    //}
}
```

3. You need to do three tasks in the OnInitialUpdate function:

   a. Construct your new recordset.
   b. Set the joined variable pointer that is used as a parameter in the second recordset to the address of the joined variable in the first recordset.
   c. Call the function to open your recordset after the primary recordset has been opened.

These three steps can be seen in the nonshaded lines that follow:

```cpp
void CODBCDepartmentView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_oDBCDepartmentSet;
    //Step 1: Construct a new CODBCInstructorSet
    m_pInstructorSet = new CODBCInstructorSet(m_pSet->m_pDatabase);
    /*
Step 2: Set the \texttt{m\_pDepartmentCode} \texttt{CODBCInstructorSet} pointer in the \texttt{m\_pInstructorSet} class to address the \texttt{m\_DepartmentCode} variable in the \texttt{m\_oODBCDepartmentSet} class

```cpp
m\_pInstructorSet->m\_pDepartmentCode =
&GetDocument() ->
\texttt{m\_oODBCDepartmentSet.m\_DepartmentCode};
```

CRecordView::OnInitialUpdate();
GetParentFrame()->RecalcLayout();
ResizeParentToFit();
m\_pSet->Close();  //Close to start transaction
if (m\_pSet->m\_pDatabase->CanTransact()) {
    m\_pSet->m\_pDatabase->BeginTrans();
}
m\_pSet->Open();  //Open after starting transaction
m\_pSet->MoveFirst();

/* Step 3: Open the Instructor recordset after the joined Department record set has been opened so you can display related instructors */
OpenInstructorRecordset();
UpdateData(FALSE);  //Update dialog box fields
}

4. Whenever you change records on the primary recordset, you need to do the following:

a. Update any related records on the secondary dataset before the move takes place.

b. Requery and display the first record on the second recordset after the move takes place.

c. If there are no related records after the move, throw the second recordset in AddNew mode to see if the user wants to add a record.

This functionality can be placed inside the \texttt{OnMove} function:

```cpp
BOOL CODBCDepartmentView::OnMove(UINT nIDMoveCommand)
{
    if (m\_bAddingRecord) {  //Currently adding?
        //If so, Update the record before the move
        UpdateData(TRUE);  //Get data from dialog box
        m\_pSet->Update();  //Update data if needed
        m\_pSet->MoveLast();  //Go to the added record
        m\_bAddingRecord = FALSE;  //Reset flag
    }
}
5. At the end of the `OnFileNew` function, add a record to the primary recordset using the add function you defined earlier:

```cpp
void CODBCDepartmentView::OnFileNew()
{
    // Previous OnFileNew code goes here
    // No instructor records, so add one.
    AddInstructor();
}
```

6. Don’t forget that during transaction processing, you need to:
   a. Update your second recordset.
   b. Get the position of your second recordset.
   c. Close your second recordset.
   d. Open the new transaction with a `BeginTrans` function, as before.
   e. Position the new transaction to the appropriate record in the recordset, if possible.

These steps are done in the `OnFileSave` function. Note that the `OpenInstructorRecordset` function (written earlier in this section) is used to both open the recordset and position it to the appropriate record:

```cpp
void CODBCDepartmentView::OnFileSave()
{
    if (m_bFieldsChanged && m_pSet->CanTransact()) {
        if (!m_bAddingRecord) { // Currently adding?
            m_pSet->Edit(); // If not, set edit mode
        }
        UpdateInstructor(); // Update Instructor too
    }
```
UpdateData(TRUE);    //Get data from dialog box
m_pSet->Update();    //Update data if needed
m_bAddingRecord = FALSE;    //Reset flag
//Commit changes
m_pSet->m_pDatabase->CommitTrans();
CRecordsetStatus rStatus;        //Status variable
//Get CRecordset status
m_pSet->GetStatus(rStatus);
m_pSet->Close();    //Close to start transaction
CRecordsetStatus rStatus2;    //Status variable
//Get CRecordset status
m_pInstructorSet->GetStatus(rStatus2);
//Close second transaction
m_pInstructorSet->Close();
//Start Transaction
m_pSet->m_pDatabase->BeginTrans();
m_pSet->Open();    //Reopen CRecordset
//Restore record position
if (rStatus.m_lCurrentRecord >= 0) {
    m_pSet->SetAbsolutePosition(rStatus.m_lCurrentRecord+1);
}
m_bFieldsChanged = FALSE;    //Reset flag
OpenInstructorRecordset(
    //Reopen second instructor
    rStatus2.m_lCurrentRecord);
UpdateData(FALSE);    //Update dialog box fields
}

7. The CRecordset class automatically detects changes made after an Edit or AddNew function call. These changes are written to the database by using the Update function. When using multiple CRecordset classes, you must make sure that all AddNew and Edit function calls are made before any UpdateData(TRUE) calls are made. UpdateData(TRUE) takes data from the dialog box and places it in the bound CRecordset variables. However, any changes that may trigger a database update are not detected until after an AddNew or Edit function is called.

Be sure to use the following steps to update your database from two CRecordset classes:

a. Place the first recordset into Edit or AddNew mode:

```cpp
if (!m_bAddingRecord) { //Currently adding?
    m_pSet->Edit(); //If not, set edit mode
}
```
b. Place the second recordset into Edit or AddNew mode.

c. Call the `UpdateData(TRUE)` function:

```
UpdateData(TRUE);  //Get data from dialog box
```

d. Call the `Update` function for both `CRecordset` classes:

```
m_pInstructorSet->Update();
```

If you use the `UpdateData(TRUE)` function before both `CRecordset` classes are in either Edit or AddNew mode, the `CRecordset` won’t be able to detect your changes and therefore won’t update the database when the `Update` function is called.

8. When you exit your program, you may need to update your recordset, and you need to deallocate your new recordset. This can be done in the `OnDestroy` function, which is triggered when the window is destroyed:

```cpp
void CODBCDepartmentView::OnDestroy()
{
    //Check for changed fields and transaction ability
    if (m_pSet->CanTransact()) {
        if (m_bFieldsChanged) {
            if (AfxMessageBox("Records have been changed. Do you want to save?", MB_YESNO) == IDNO) {
                m_pSet->m_pDatabase->Rollback();
            } else {
                if (!m_bAddingRecord) { //Currently adding?
                    //If not, set edit mode
                    m_pSet->Edit();
                }
                UpdateInstructor();
                UpdateData(TRUE);  //Get data from dialog box
                m_pSet->Update();  //Update data if needed
                m_pSet->m_pDatabase->CommitTrans();
            }
        } else {
            m_pSet->m_pDatabase->CommitTrans();
        }
    }
    else {
        m_pSet->m_pDatabase->CommitTrans();
    }
}
}  
if (m_pInstructorSet != NULL) {  
m_pInstructorSet->Close();  
delete m_pInstructorSet;  
}  
CRecordView::OnDestroy();  
}

The code for the version of ODBCDepartment can be found in the ODBCDepartmentD directory on the CD-ROM accompanying this book.

### Summary

This chapter showed how easy ODBC development can be using Visual C++ and the MFC Application Wizard. To recap:

- You can quickly generate a scrolling or updateable application with little or no code using the MFC Application Wizard.
- With a little code you can add insert and update capability and transaction support to your MFC application.
- Transactions are a little tricky, and you should check with your DBA or database documentation for any transaction restrictions on your database.
- Joins are almost as easy as single table applications when using the MFC Application Wizard.
- Sometimes, your application may require multiple CRecordset objects. The easiest way to do this is to start with an MFC application and add additional functionality.