Lab 9.1: Building a Database Editor with ODBC

Objectives

After completing this lab, you will be able to:

- Use AppWizard to create an ODBC database application.
- Connect controls in a CRecordView to their CRecordset variables.
- Create a parameterized query filter for data retrieval.
- Process CRecordView::OnMove for multiple-table record coordination, update, deletion, and addition.
- Determine whether an ODBC dataset supports transactions.
- Use transactions in a program.

Prerequisites

You should have completed Chapter 9 and mastered the use of the dialog editor before attempting this lab.

Lab Setup

This demonstration shows what you will accomplish during the lab.

Estimated time to complete this lab: **90 minutes**.

Exercises

The following exercise provides practice working with the concepts and techniques covered in this chapter.

Exercise 1: Building an ODBC Database Application

In this exercise, you implement a two-table ODBC database application.

Exercise 2: Editing the Underlying Database

In this exercise, you enable the Employee database application to:

- Add a new record.
- Modify a record.
- Delete a record.

Before starting this lab, use the 32-bit ODBC driver manager in Control Panel to add a data source using the Microsoft Access Driver. Name the data source **PERSONAL**. Use the Personnel.mdb database located in \Labs\C09\Lab01.

Copy the file \Labs\C09\Lab01\EmployeeDdx.cpp to your lab directory. This file contains custom DDX and DDV functions for calendar-date data.

There is no setup for this lab. The completed code for these exercises is in \Labs\C09\Lab01\Xxx where Xxx is the exercise number.

Exercise 1: Building an ODBC Database Application

In this exercise, you will implement a two-table ODBC database application.

The database for this lab, Personnel.Mdb, contains two tables: the Employee Pay Table and Employee Personal Info Table.

This is the schema for the Employee Pay Table.

Table: Employee Pay Table

Field Name	Data Type	Description
Employee Number	Number	Primary Index - Employee Number
Last Name	Text	Employee's Last Name (28 char max)
First Name	Text	Employee's First Name (28 char max)
Department #	Number	0-unassigned, 1-manufacturing, 2-administration, 3-information
Employee Pay Type	Number	0-Unknown, 1-Hourly Employee, 2-Salaried, 3-Salesperson
Hours	Number	Hourly employee's weekly hours
Hourly Rate	Number	Hourly employee's hourly wage
Weekly Salary	Number	Salaried employee's weekly salary
Sales Bonus Rate	Number	Salaesperson's bonus rate
Weekly Sales	Number	Salaesperson's weekly total sales

This is the schema for the Employee Personal Info Table.

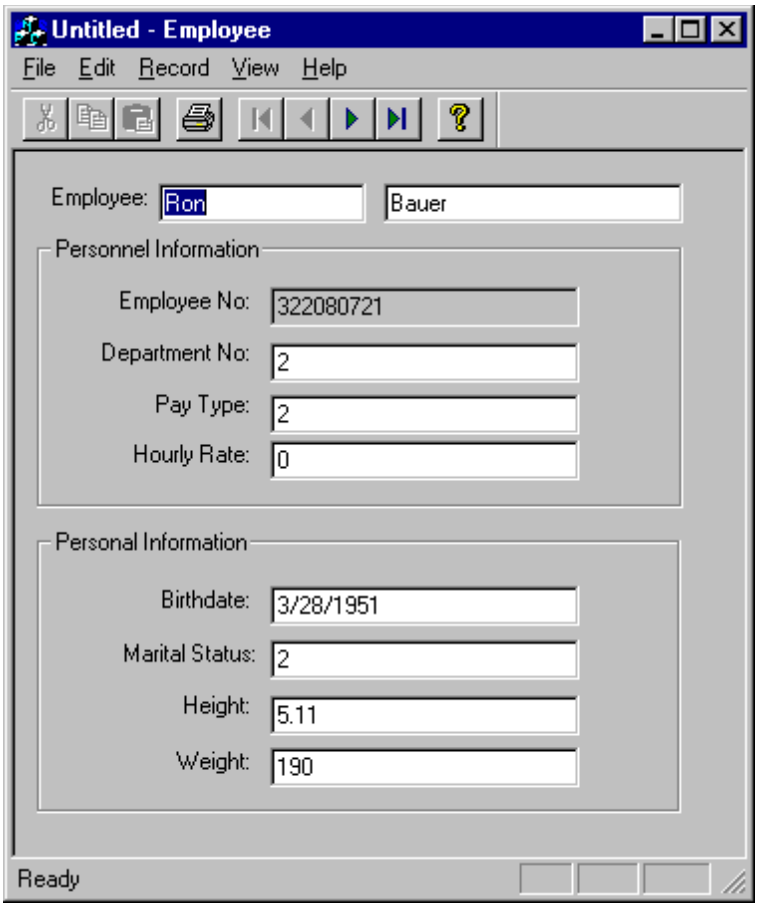
Table: Employee Personal Info Table

Field Name	Data Type	Description
Employee Number	Number	Primary Index - Employee Number
Birthdate	Date/Time	Employee's date of birth
Sex & Marital Status	Number	1-single male, 2-married male, 3-single female, 4-married female
Height	Number	Employee's height
Weight	Number	Employee's weight

Employee Number is the common key between the two tables. The SQL query to synchronize the recordsets is:

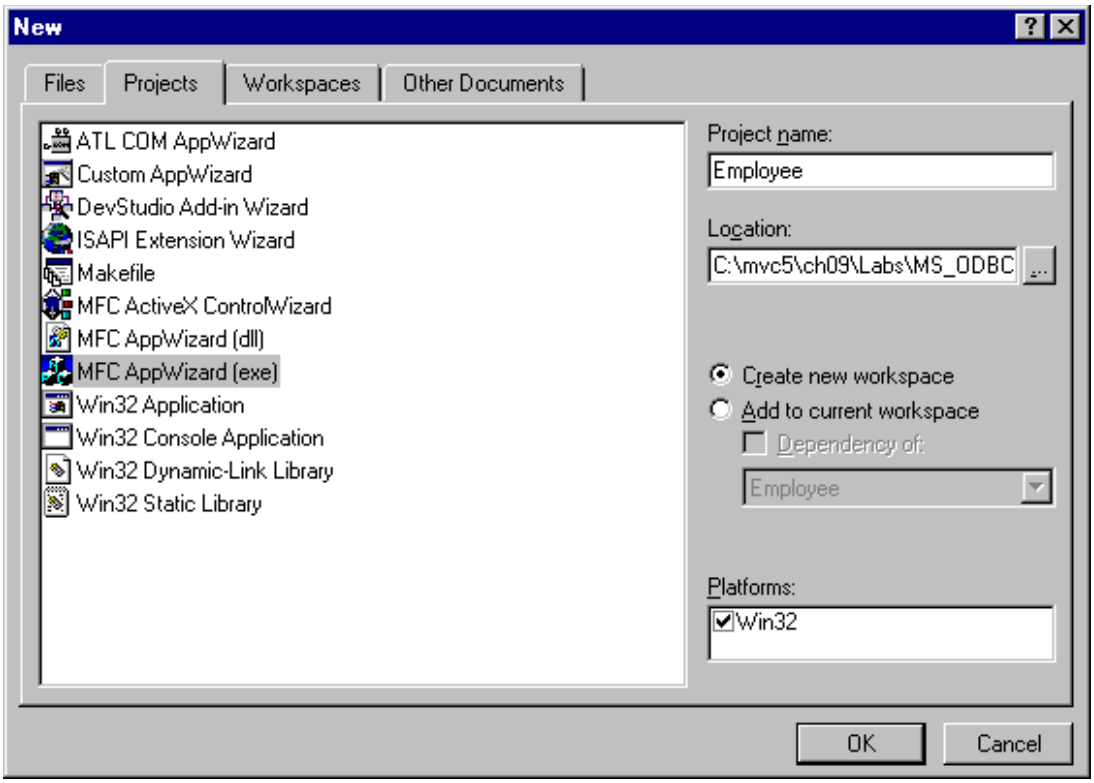
```
Select *
    from [Employee Personal Info Table], [Employee Pay Table]
    where [Employee Personal Info Table].[Employee Number] =
          [Employee Pay Table].[Employee Number]
```

Your complete application will look like the following graphic.

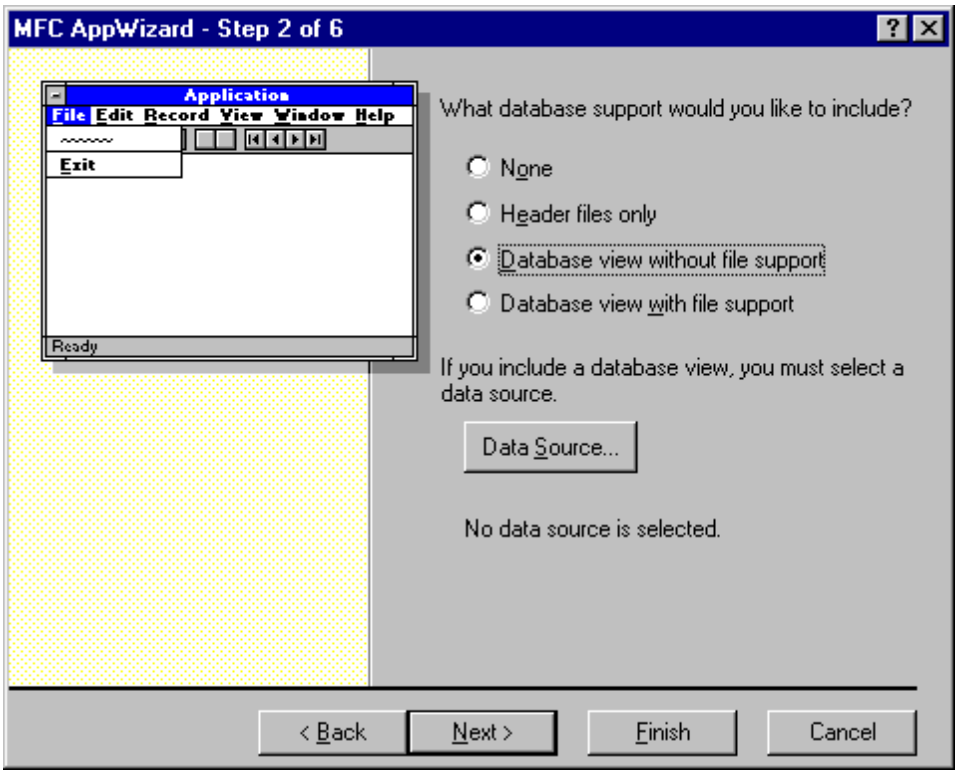


➢ **Create a new ODBC database application**

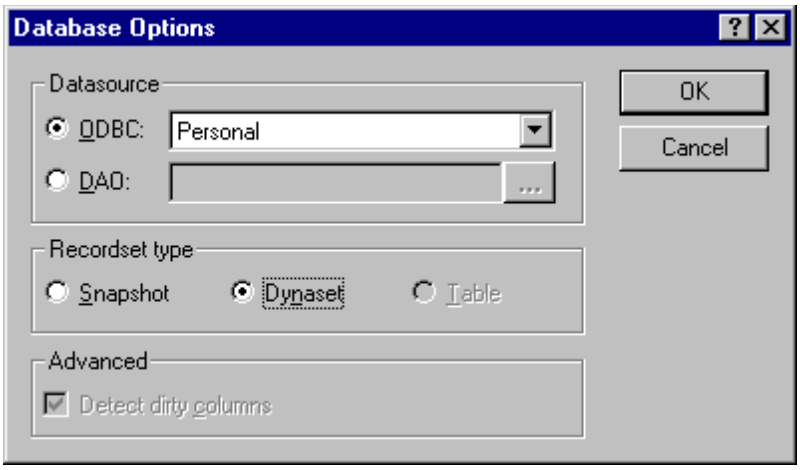
1. From the File menu, choose New.
2. Select the Projects tab; choose MFC AppWizard (exe), and type **Employee** in the Project Name box. Click OK.



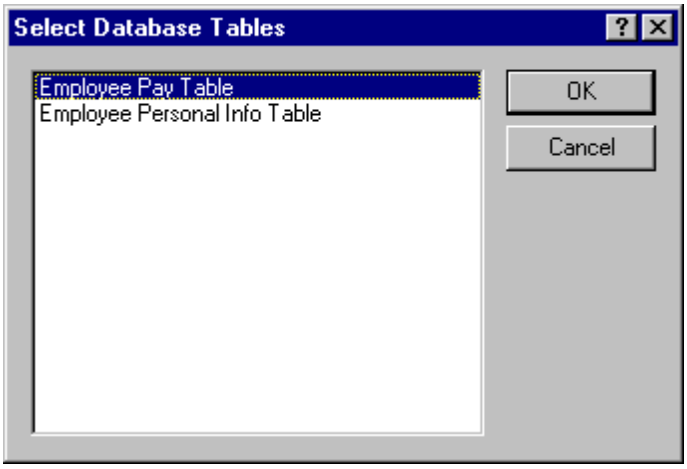
3. In Step 1, choose Single Document Interface.
4. In Step 2 (the Database Options Page), select the “Database view without file support” option.



5. Click Data Source. The Database options page will be displayed. Choose the ODBC option. Choose PERSONAL from the list box. Choose Dynaset as the Resource type.



When you click OK, the Select Database Tables dialog will be displayed. Select the Employee Pay Table and click OK. You will be returned to the Database Options page.



6. Accept the defaults in Steps 3, 4, and 5.
7. In Step 6, notice the base classes for the various classes in the application:

Class	Base Class
CEmployeeApp	CWinApp
CEmployeeDoc	CDocument
CMainFrame	CFrameWnd
CEmployeeView	CRecordView
CEmployeeSet	CRecordset

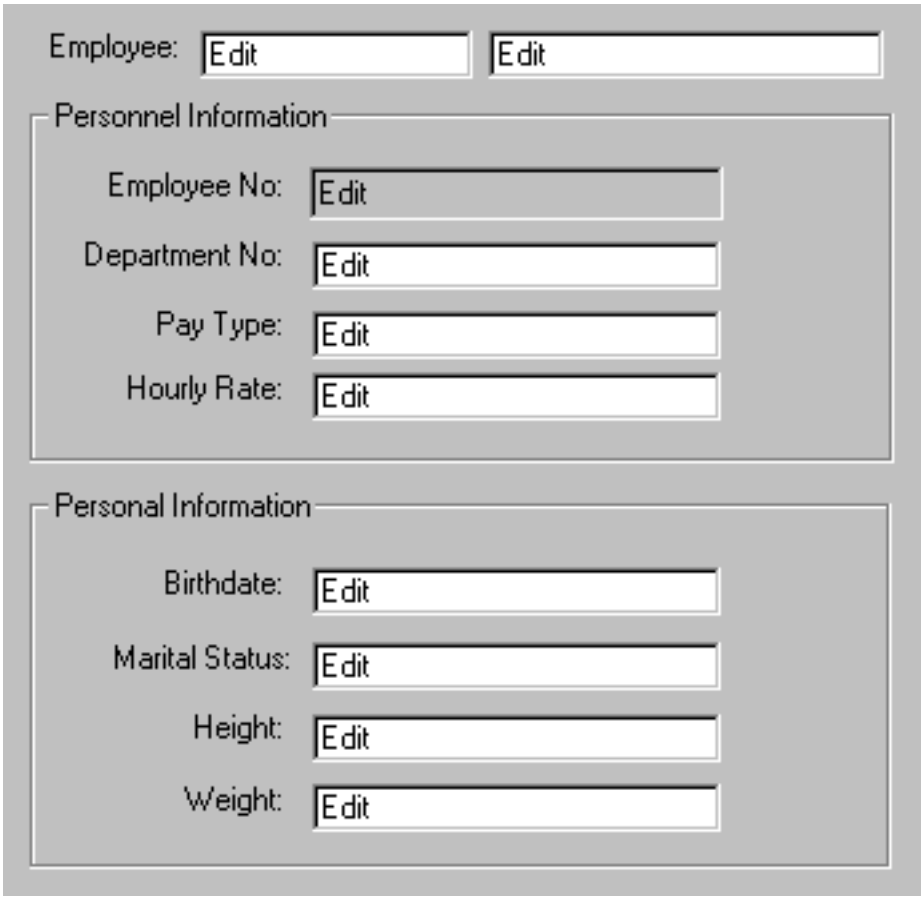
8. Change the name of **CEmployeeSet** to **CEmployeePaySet**. You will be creating another **CRecordset**-derived class for the other table in the database soon, and you will need to be able to tell them apart. Change the files for **CEmployeePaySet** to EmployeePaySet.H and EmployeePaySet.Cpp.
9. Click OK in the Project Information dialog to create the project.

➢ **Customize the Dialog Template for the Employee form**

1. Because **CEmployeeView** is derived from **CRecordView**, and **CRecordView** is derived from **CFormView**, you will use a dialog template to define the client area. Along with the classes it created, AppWizard created a dialog resource, IDD_EMPLOYEE_FORM, for you to lay out. AppWizard places

one static text control in the resource, labeled “TO DO: Place form controls on this dialog.” Open IDD_EMPLOYEE _FORM by clicking this dialog’s icon in the Resource View.

2. Delete the one static control. Enlarge the dialog box to a width of 245 and a height of 220 so there will be room for the controls you are going to copy. Now copy the controls from the solution: Open the solution file \XXX\Ex01\Employee.Rc. Open the dialog resource IDD_EMPLOYEE_FORM in the solution file. Use the DevStudio menu to choose Edit Select All. Choose Edit Copy. Close the solution Rc file. Click the application’s dialog form, press Edit Paste.



The following table shows the controls and their IDs.

Type	ID	Caption
Dialog	IDD_EMPLOYEE_FORM	
Right aligned text	IDC_STATIC	Employee:
Right aligned text	IDC_STATIC	Employee No:
Right aligned text	IDC_STATIC	Department No:
Right aligned text	IDC_STATIC	Pay Type:
Right aligned text	IDC_STATIC	Hourly Rate:
Right aligned text	IDC_STATIC	Birthdate:
Right aligned text	IDC_STATIC	Marital Status:
Right aligned text	IDC_STATIC	Height:
Right aligned text	IDC_STATIC	Weight:
Group box	IDC_STATIC	Personnel Information
Group box	IDC_STATIC	Personal

		Information
Edit control	IDC_FNAME	First Name
Edit control	IDC_LNAME	Last Name
Edit control	IDC_EMP_NO	**Note:** This control is read-only
Edit control	IDC_DEPT	
Edit control	IDC_PAY_TYPE	
Edit control	IDC_HOUR_RATE	
Edit control	IDC_BIRTH	
Edit control	IDC_MARITAL_STATUS	
Edit control	IDC_HEIGHT	
Edit control	IDC_WEIGHT	

3. Save Employee.Rc.

➢ **Bind controls to member variables**

You can use an extension to the dialog editor, or the ClassWizard, to bind controls to the appropriate member variables of **CEmployeePaySet**, according to the following table. You will bind the last four Edit controls after you have created a **CRecordset** for their table.

Control	Data Type	Variable
IDC_FNAME	CString	m_pSet->m_First_Name
IDC_LNAME	CString	m_pSet->m_Last_Name
IDC_EMP_NO	long	m_pSet->m_Employee_Number
IDC_DEPT	int	m_pSet->m_Department__
IDC_PAY_TYPE	int	m_pSet->m_Employee_Pay_Type
IDC_HOUR_RATE	double	m_pSet->m_Hourly_Rate

1. To bind using the Dialog editor, CTRL+double-click the control to display the Add Member Variable dialog. This step is easiest if you first set the tab order of all controls sequentially, with each Static Text control immediately preceding its associated Edit Box in the tab number sequence. The wizard tries to match the Caption of the Static control with a recordset member name. The Static control must preceed the Edit control in the tab order for matching to occur.
2. To start ClassWizard, use the View menu or press CTRL+W, then click the Member Variables tab. Choose **CEmployeeView** from the Class name list box, and browse the Control IDs. Click Add Variable to display the Add Member Variable dialog box.

   Use the Member variable name combo box to select the variable name. Leave the category as Value, and the Variable type should default to the data types listed above.
3. Close the ClassWizard or the Dialog editor and save all files.

➢ **Order the display of the records**

1. In ODBC, you can specify the ORDER BY clause in m_strSort. Add this to the end of the **CEmployeePaySet** constructor.

```
m_strSort = "[Last Name],[First Name]";
```

2. Save EmployeePaySet.Cpp.

➢ **Create a class for a second table and bind its variables**

1. Start the ClassWizard from the View menu or press CTRL+W. Click Add Class, then click New.
2. Set the name of this new class to **CEmployeePersonalSet**. Set its base class to **CRecordset**, and click OK.

3. The Database Options dialog that you used to create **CEmployeePaySet** will be displayed, with one additional option: Bind all columns. Make sure that this option is checked. Select the dynaset recordset type. Leave the ODBC option selected and choose the PERSONAL data source name. Click OK.
4. In the Select Database Tables dialog box, choose Employee Personal Info Table and click OK. **CEmployeePersonalSet** will be created and added to your project.
5. Select the wizard's Member Variables tab. Delete the CTime variable m_Birthdate. The Class Wizard binds to a CTime variable, which is not the correct type for our needs. Manually add the variable to the **public** section of the class **CEmployeePersonalSet**:

```
TIMESTAMP_STRUCT m_Birthdate;
```

6. Assign zero to each data-member of m_Birthdate, in the class constructor.

```
m_Birthdate.year    = 0;
m_Birthdate.month   = 0;
m_Birthdate.day     = 0;
m_Birthdate.hour    = 0;
m_Birthdate.minute  = 0;
m_Birthdate.second  = 0;
m_Birthdate.fraction= 0;
```

7. Modify the class member function, **CEmployeePersonalSet::DoFieldExchange**, to transfer data to m_Birthdate. Place the code after the section //}}AFX_FIELD_MAP

```
RFX_Date(pFX, _T("[Birthdate]"), m_Birthdate);
```

8. In the class constructor, find the section //{{AFX_FIELD_INIT(CEmployeePersonalSet). Manually increment the data member to indicate the additional field transfer:

```
m_nFields = 5;
```

9. Add a **CEmployeePersonalSet** instance variable to the **CEmployeeView** class immediately following the AFX_DATA section of EmployeeView.H.

```
public:
//{{AFX_DATA(CEmployeeView)
enum { IDD = IDD_EMPLOYEE_FORM };
CEmployeePaySet*    m_pSet;
//}}AFX_DATA
CEmployeePersonalSet*   m_pEmplInfoSet;
```

10. In the constructor for **CEmployeeView**, initialize m_pEmplInfoSet to NULL.

```
m_pEmplInfoSet = NULL;
```

11. At the top of EmployeeView.H, add the following forward declaration:

```
class CEmployeePersonalSet;
```

➢ **Modify DoDataExchange to transfer data from the second table**

1. You need to manually edit the DDX links. Following the AFX_DATA_MAP section of **CEmployeeView::DoDataExchange**, validate m_pEmplInfoSet and then set the following DDX links. Be sure to use the correct recordset pointer. The DDX_FieldText function for the Birthdate field is an overloaded version provided for you in EmployeeDdx.Cpp.

Control	Data type	Variable
IDC_BIRTH	TIMESTAMP_STRUCT	m_pEmplInfoSet ->m_Birthdate
IDC_MARITAL_STATUS	BYTE	m_pEmplInfoSet ->m_Sex___Marital_Status

IDC_HEIGHT	double	m_pEmplInfoSet ->m_Height
IDC_WEIGHT	double	m_pEmplInfoSet ->m_Weight

2. Immediately following the data exchange call for m_Birthdate, enter the following call to DDV_Date. This is an overloaded version provided for you in EmployeeDdx.Cpp:

```
DDV_Date( pDX, m_pEmplInfoSet->m_Birthdate, m_pEmplInfoSet );
```

3. At the top of EmployeeView.Cpp, following the #include directives, place the following:

```
#include "EmployeeDdx.Cpp"
```

4. Save EmployeeView.Cpp. The complete function follows.

```
void CEmployeeView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CEmployeeView)
    DDX_FieldText(pDX, IDC_FNAME, m_pSet->m_First_Name, m_pSet);
    DDX_FieldText(pDX, IDC_LNAME, m_pSet->m_Last_Name, m_pSet);
    DDX_FieldText(pDX, IDC_EMP_NO, m_pSet->m_Employee_Number, m_pSet);
    DDX_FieldText(pDX, IDC_DEPT, m_pSet->m_Department__, m_pSet);
    DDX_FieldText(pDX, IDC_PAY_TYPE, m_pSet->m_Employee_Pay_Type, m_pSet);
    DDX_FieldText(pDX, IDC_HOUR_RATE, m_pSet->m_Hourly_Rate, m_pSet);
    //}}AFX_DATA_MAP

    //Ensure pointer is valid
    if ( NULL != m_pEmplInfoSet )
    {
        DDX_FieldText(pDX, IDC_BIRTH, m_pEmplInfoSet->m_Birthdate,
m_pEmplInfoSet);
        DDV_Date( pDX, m_pEmplInfoSet->m_Birthdate, m_pEmplInfoSet );
        DDX_FieldText(pDX, IDC_MARITAL_STATUS, m_pEmplInfoSet-
>m_Sex___Marital_Status, m_pEmplInfoSet);
        DDX_FieldText(pDX, IDC_HEIGHT, m_pEmplInfoSet->m_Height,
m_pEmplInfoSet);
        DDX_FieldText(pDX, IDC_WEIGHT, m_pEmplInfoSet->m_Weight,
m_pEmplInfoSet);
    }
}
```

5. Embed a protected **CDatabase** member into **CEmployeeDoc**. Name the member m_DB.

```
CDatabase m_DB;
```

6. Embed **CEmployeePersonalSet** into **CEmployeeDoc** by adding an instance of **CEmployeePersonalSet** to EmployeeDoc.H.

```
CEmployeePersonalSetm_employeePersonalInfoSet;
```

Note The declaration for **CDatabase** m_DB, done in Step 5, *must* precede the declaration of the **CEmployeePersonalSet** and **CEmployeePaySet** members, because the **CDatabase** object must be constructed before the two **CRecordset**-derived objects.

---

7. Add accessor functions for both the **CRecordset** objects.

```
// Operations
public:
CEmployeePaySet*    GetEmployeePaySet()
{return &m_employeePaySet;}
```

```
CEmployeePersonalSet*   GetEmployeeInfoSet()
{return &m_employeePersonalInfoSet;}
```

8. Add colon initialization to the document constructor **CEmployeeDoc::CEmployeeDoc** to initialize the recordset members with the database.

```
CEmployeeDoc::CEmployeeDoc( )
    : m_employeePaySet( & m_DB ), m_employeePersonalInfoSet( & m_DB )
{
    …
```

9. Include EmployeePersonalSet.H in EmployeeView.Cpp, EmployeeDoc.Cpp, and in Employee.Cpp. In each case, insert the include directive immediately before the directive that includes EmployeePaySet.H.
10. To avoid multiple inclusions, wrap the recordset header-files with preprocessor directives. In EmployeePaySet.H, insert these two lines at the top:

```
#ifndef _CEmployeePaySet_H
#define _CEmployeePaySet_H
```

    and at the bottom of the file, insert

```
#endif
```

11. In EmployeePersonalSet.H, insert these two lines at the top:

```
#ifndef _CEmployeePersonalSet_H
#define _CEmployeePersonalSet_H
```

    and at the bottom of the file, insert

```
#endif
```

12. Save all files.

➢ **Program the tables' relationship**

Not all databases support updatable joins. Your program has to maintain the relationship between the two tables. One way is to create a parameterized query that finds the record in Employee Personal Information Table that corresponds to the record found in the Employee Pay Table.

There are two methods for creating a parameterized query. You can use the **PARAMETERS** statement in the SQL query (as outlined in Chapter 10), or you can update the recordset class as shown in the following method.

1. Declare a **long** for the parameter in **CEmployeePersonalSet** after the AFX_FIELD block.

```
long m_EmployeeNumberParam;
```

2. Save EmployeePersonalSet.H. In the constructor for **CEmployeePersonalSet**, define the query. Note that the question mark serves as the parameter placeholder.

```
m_strFilter = "[Employee Number] = ?";
```

3. Indicate that there is one parameter. Initialize that parameter to 0.

```
m_nParams = 1;
m_EmployeeNumberParam = 0;
```

4. In **CEmployeePersonalSet::DoFieldExchange**, you need to identify the query as parameterized. Before the AFX_FIELD_MAP, use SetFieldType to parameterize the recordset.

```
pFX->SetFieldType(CFieldExchange::param);
```

5. Bind the parameterized field to the variable.

```
RFX_Long(pFX, _T("EmployeeNumberParam"), m_EmployeeNumberParam);
```

6. Save EmployeePersonalSet.Cpp.

➢ **Implement CEmployeeView::OnInitialUpdate**

1. Before the **CRecordView::OnInitialUpdate** statement, set the instance variable for **CEmployeePersonalSet**.

```
m_pEmplInfoSet = GetDocument( )->GetEmployeeInfoSet( );
```

2. Resize the main window so that it matches the dialog template.

```
GetParentFrame()->RecalcLayout();
ResizeParentToFit(FALSE);
```

3. Start a **try** block. Open the primary table. Because **CRecordset::Open** can throw exceptions, you place this call in a **try** block.

```
try
{
        m_pSet->Open( );
```

4. Set the employee number parameter for the **CEmployeePersonalSet** filter from the employee number field of **CEmployeePaySet**.

```
m_pEmplInfoSet->m_EmployeeNumberParam =
    m_pSet->m_Employee_Number;
```

5. Open the secondary table, and close the **try** block.

```
        m_pEmplInfoSet->Open( );
    }
```

6. Write a **catch** block to trap any exception thrown by **CRecordset::Open**. Report the error to the user.

```
catch (CException * pEx)
{
    pEx->ReportError( );
```

7. Delete the exception and return from the function without updating the display. Close the **try** block.

```
pEx->Delete( );
return;
}
```

8. Use DDX to update the screen display from the view's member variables.

```
UpdateData(FALSE);
```

9. Save EmployeeView.Cpp.

➢ **Implement CEmployeeView::OnMove to synchronize recordsets**

The virtual function **OnMove** is called from the default functions **OnRecordFirst**. **OnRecordPrevious**, **OnRecordLast**, and **OnRecordNext** of the **CRecordView** class. In the simplest case, the call goes to the base class **OnMove** function. In this lab, you override **OnMove** to force a requery of **CEmployeePersonalSet** after a move of the primary set, **CEmployeeSet**.

1. Use ClassWizard or the WizardBar to create an override of **CRecordView::OnMove**. The function created by ClassWizard includes a call to the base class **OnMove**. To save any user changes to the dialog controls, precede the call to the base class with a call to **UpdateData**. If **UpdateData** returns FALSE, immediately return FALSE from the function.

```
BOOL CEmployeeView::OnMove(UINT nIDMoveCommand)
{
    if ( ! UpdateData( TRUE ) ) //If DDX failed
```

```
        return FALSE;   //Bad data: We don't want to move
```

2. Call **CRecordset:: SetFieldDirty** to notify the base class we want the data saved.

```
m_pSet->SetFieldDirty( NULL );
        //Otherwise DB update won't occur
```

3. Remove the **return** statement from the call to the base class. Define a **BOOL** variable to store the returned value.

```
BOOL bMove = CRecordView::OnMove(nIDMoveCommand);
        //Primary table updated
```

4. Write an **if** control block based on the Boolean variable.

```
if ( bMove )//If successful update the foreign info
{
```

5. To save any user changes, put the foreign recordset into edit mode, mark all fields as dirty, and then save the data.

```
        m_pEmplInfoSet->Edit( );//Turn on edit mode
        m_pEmplInfoSet->SetFieldDirty( NULL );
                                    //Mark the fields dirty
        m_pEmplInfoSet->Update( );
```

6. Set the parameter for **CEmployeePersonalSet** from the primary recordset's employee number.

```
        m_pEmplInfoSet->m_EmployeeNumberParam =
                m_pSet->m_Employee_Number;
```

7. Execute the query on **CEmployeePersonalSet**.

```
        m_pEmplInfoSet->Requery();
```

8. Initiate a data transfer to the dialog, then close the **if** block.

```
        UpdateData(FALSE);
}
```

9. Return the Boolean's value to complete the function.

```
return bMove;
```

10. Save EmployeeView.Cpp.

➢ **Build and run Employee.Exe**

Your application has the same functionality as Join.exe from Lab 8.1, except that you can modify data in the edit controls. Employee Number remains read-only.

The completed code for this exercise is in \Labs\C09\Lab01\Ex01.

Exercise 2: Editing the Underlying Database

Continue with the files you created in Exercise 1 or, if you do not have a starting point for this exercise, the code that forms the basis for this exercise is in \Labs\C09\Lab01\Ex01.

In this exercise, you will edit the Employee project in order to enable the user to do the following:

- Add a new record.
- Delete a record.
- Use transactions when supported.

There are four parts to this exercise:

1. Adding items to the menu.
2. Preparing to add a record.
3. Adding a record.
4. Deleting a record.

Adding Items to the Menu

➢ **Copy menu resources from the lab solution**

1. Open the file \Labs\C09\Lab01\Ex02\Employee.Rc. Expand the menu tree. Choose IDR_MAINFRAME. Single-click the menu-selection “&Record." Press CTRL+C to copy the menu selection. Close the file.
2. Open the IDR_MAINFRAME menu for this application. Single-click the menu-selection “&Record." Press CTRL+V to paste the menu selection copied from the solution file. This adds a separator to the end of the Record menu, and adds the following items to the Record menu:

ID	Caption	Prompt
ID_RECORD_CLEAR	&Clear Record	Clear the fields in the form
ID_RECORD_ADD	&Add Record	Add this record to the database
ID_RECORD_DELETE	&Delete Record	Delete this record from the database

3. Save Employee.Rc.
4. Use the ClassWizard to create command handlers in **CEmployeeView** for each of the three menu items.
5. Create an Update Command UI handler for ID_RECORD_DELETE.
6. Create an Update Command UI handler for ID_RECORD_ADD.

Preparing to Add Records

The first step to add a record to a database is to set the recordset(s) to **AddNew** mode. **AddNew** prepares an empty record using the recordset’s field data members. Taken collectively, the field data members of a recordset serve as an “edit buffer” that contains one record•in this case, a new record. If the user moves to another record before saving the new record, your application will cancel the addition, and return to normal viewing. You will program the functions to use transactions when the database supports transactions.

➢ **Implement OnRecordClear to prepare the recordsets and the dialog**

1. In the ClassView pane, expand the **CEmployeeView** class, and double-click the handler function **OnRecordClear** to edit the code.
2. On the line after the opening brace of this function, retrieve and store the **CDatabase** pointer from the primary recordset.

```
CDatabase * pDB = m_pSet->m_pDatabase;
```

3. **CRecordset** does not support transactions across databases. ASSERT that the primary and secondary recordsets point to the same database.

```
ASSERT ( m_pEmplInfoSet->m_pDatabase == pDB );
```

4. If **CView::IsAddMode** returns TRUE (you will program this function in a moment), call **CancelUpdate** for each recordset.

```
if ( IsAddMode( ) )
{
    m_pSet->CancelUpdate( );
```

```
    m_pEmplInfoSet->CancelUpdate( );
}
```

5. If add mode is inactive, determine whether the database supports transactions. (Use an **else if** block.)

```
else if ( pDB->CanTransact( ) )
{
```

6. Within the **else if** block, call **CDataBase::GetCursorCommitBehavior** to determine whether to start the transaction with an open cursor (recordset) or to close the recordsets and then start the transaction. Close all code blocks.

```
    if ( SQL_CB_PRESERVE == pDB->GetCursorCommitBehavior( ) )
        pDB->BeginTrans( );
    else//Cannot have an open recordset to start transaction
    {
        m_pSet->Close( );
        m_pEmplInfoSet->Close( );
        pDB->BeginTrans( );
        m_pSet->Open( );
        m_pEmplInfoSet->Open( );
    }
}   //End can transact
```

Note For further information about transactions, see Technical Note 68, "Performing Transactions with the Microsoft Access 7 ODBC Driver" in the Microsoft Foundation Class Reference, which is part of the Microsoft Visual C++ online documentation.

---

7. Call **CRecordset::AddNew** for each recordset.

```
m_pSet->AddNew( );
m_pEmplInfoSet->AddNew( );
```

8. Call **CView::SetAddMode** (programmed in the next step) and **CView::UpdateData** to clear the dialog controls.

```
SetAddMode( );
UpdateData( FALSE );
```

9. Save your work.

➢ **Implement SetAddMode and IsAddMode**

1. In the ClassView pane, right-click **CEmployeeView**, choose Add Member Function. Type **void** in the Return type box and type **SetAddMode(BOOL bAddMode = TRUE)** in the Declaration box.
2. In the ClassView pane, right-click **CEmployeeView**, choose Add Member Function. Type **BOOL** in the Return type box and type **IsAddMode** in the Declaration box.
3. In the ClassView pane, right-click **CEmployeeView**, choose Add Member Variable. Type **m_bAddMode** in the Name box and choose Protected for the access mode. the data type is **BOOL**.
4. Open EmployeeView.Cpp. In the body of the **SetAddMode** function, set the add mode state to the passed value.

```
m_bAddMode = bAddMode;
```

5. Get a pointer to the employee number edit control.

```
CEdit * pField = ( CEdit * ) GetDlgItem( IDC_EMP_NO );
```

6. Set the state of that edit control to readable when you are in add mode, or read-only when you are not in add mode.

```
pField->SetReadOnly( ! m_bAddMode );
```

7. If function argument is TRUE, set the focus to the first-name edit control.

```
if( m_bAddMode )
{
    pField = ( CEdit * )GetDlgItem( IDC_FNAME );
    pField->SetFocus( );
}
```

8. Program **IsAddMode** to return the value **m_bAddMode**.
9. Save EmployeeView.Cpp.

➢ **Cancel Add Mode**

1. In the ClassView pane, right-click **CEmployeeView**, choose Add Member Function. Type **void** in the Return type box and type **AddRecordCancel** in the Declaration box.
2. Open EmployeeView.Cpp. In the body of **AddRecordCancel** function, add code to **return** if not in add mode.

```
if ( ! m_bAddMode )
    return;
```

3. Turn off add mode.

```
SetAddMode( FALSE );
```

4. Cancel the update on both tables.

```
m_pSet->CancelUpdate( );
m_pEmplInfoSet->CancelUpdate( );
```

➢ **Roll back a pending transaction**

1. Continue adding code to the body of AddRecordCancel to get a pointer to the database, and then determine whether the database supports transactions.

```
CDatabase * pDB = m_pSet->m_pDatabase;
if ( pDB->CanTransact( ) )
{
```

2. Roll back the transaction.

```
    pDB->Rollback( );
```

3. Some databases require that you close the cursor(s) after a rollback. If required, you need to do so.

```
    if ( SQL_CB_DELETE == pDB->GetCursorRollbackBehavior( ) )
    {
        m_pSet->Close( );
        m_pEmplInfoSet->Close( );
        m_pSet->Open( );
        m_pEmplInfoSet->Open( );
    }
```

4. Requery each table.

```
    m_pSet->Requery( );
    m_pEmplInfoSet->m_EmployeeNumberParam =
    m_pSet->m_Employee_Number;m_pEmplInfoSet->Requery( );
}
```

5. Refresh the form.

```
UpdateData( FALSE );
```

6. Save EmployeeView.Cpp.

➢ **Implement the Command UI handlers**

1. Enable the Add menu item only when you are in add mode.

```
void CEmployeeView::OnUpdateRecordAdd(CCmdUI* pCmdUI)
{
    pCmdUI->Enable( IsAddMode( ) );

}
```

2. Enable the Delete menu item only when you are not in add mode.

```
void CEmployeeView::OnUpdateRecordDelete(CCmdUI* pCmdUI)
{
    pCmdUI->Enable( ! IsAddMode( ) );

}
```

3. Set add mode to **FALSE** in **CEmployeeView::OnInitialUpdate** before the call to the base class.

```
SetAddMode( FALSE );
CRecordView::OnInitialUpdate( );
```

4. Save EmployeeView.Cpp.

➢ **Reset Add Mode on navigation**

The default handler for **CRecordView::OnMove** updates a record before moving off the record. We want a move to cancel AddMode, if AddMode is set.

1. To cancel AddMode, as the first statement in **OnMove**, insert a call to the **AddRecordCancel** function, which you created earlier in this exercise.

```
if ( IsAddMode( ) )
    AddRecordCancel( );
```

2. Save EmployeeView.Cpp.

Adding a Record

Whether adding or updating records, changes need synchronization between tables. **OnRecordAdd** will use transactions when available. Transactions encapsulate database modifications to help maintain data integrity.

➢ **Add a record**

1. In **CEmployeeView:: OnRecordAdd**, get and store a database pointer. Determine whether the database supports transactions. Save the information.

```
CDatabase * pDB = m_pSet->m_pDatabase;
BOOL bTrans = pDB->CanTransact( );
```

2. Update the recordsets from the form. If the update is successful, execute a **try** block.

```
if ( UpdateData( TRUE ) )    //Scrape the screen data into memory
    try
    {
```

3. Within the try block, first update the secondary table's employee number.

```
        m_pEmplInfoSet->m_Employee_Number =
            m_pSet->m_Employee_Number;
```

4. Update each record in its corresponding table.

```
        m_pSet->Update( );
```

```
        m_pEmplInfoSet->Update( );
```

5. If the database supports transactions, commit the transaction.

```
        if ( bTrans )
        {
            BOOL bCommitOk = pDB->CommitTrans( );
```

6. If **CommitTrans** failed, or if the database does not preserve the cursor, our recordsets might be in indetermined states. In either case, we should **Close** and then **Open** the recordsets.

```
            if ( ! bCommitOk ||
                SQL_CB_PRESERVE != pDB->GetCursorCommitBehavior( ) )
                                    //See TN068
            {
                m_pSet->Close( );
                m_pEmplInfoSet->Close( );
                m_pSet->Open( );
                m_pEmplInfoSet->Open( );
            }
```

7. Close the outer **if** block. Call **SetAddMode** with an argument of FALSE.

```
        }
        SetAddMode( FALSE );
```

8. Call OnRecordClear to prepare for the next new record. Close the **try** block.

```
        OnRecordClear( );   //Set up to add the next record
    }   //End try
```

➢ **Handle an exception during OnRecordAdd**

If an exception occurs without an active transaction, **CRecordset** leaves the recordset in AddNew mode. An active transaction, however, may require closing the recordset after a rollback, or the transaction may leave the recordset mode in an undetermined state.

Our logic is to assume the worst with a pending transaction. We will close the recordset and then reset the mode. Since the current record is an empty edit buffer during AddNew mode, and since the dialog controls contain the user's input data, everything works out nicely.

1. In the ClassView pane, expand the **CEmployeeView** class, and double-click the handler function **OnRecordAdd** to edit the code.
2. On the line after the opening brace of this function, if an exception occurs, report the error, then delete the exception.

```
    catch ( CDBException * pEx )
    {
        pEx->ReportError( );
        pEx->Delete( );
```

3. If a transaction is pending, roll back the transaction.

```
        if ( bTrans )
        {
            pDB->Rollback( );
```

4. Assume the worst behavior from the recordsets, and close them both.

```
            m_pSet->Close( );   //Assume worst case
            m_pEmplInfoSet->Close( );
```

5. Begin a transaction (again), open the recordsets, and reset the recordsets' modes. Close all code blocks.

```
            pDB->BeginTrans( ); //Start trans, again
            m_pSet->Open( );
            m_pEmplInfoSet->Open( );

            m_pSet->AddNew( );
            m_pEmplInfoSet->AddNew( );
        }   //End if bTrans
    }   //End catch
}   //End if UpdateData(TRUE)
```

6. Save EmployeeView.Cpp.

Deleting a Record

You delete two related records in much the same way that you add and update them. You will start a transaction (when supported), delete the records and, if the transaction fails, recover. To simplify programming, we assume that the recordsets must be closed after a **CommitTrans** or a **Rollback**. In a production environment, you will want to query for the recordset's behavior, as in earlier exercises. Repeatedly opening and closing recordsets is inefficient.

➢ **Delete a record**

1. In the ClassView pane expand the **CEmployeeView** class, and double-click the handler function **OnRecordDelete** to edit the code.
2. On the line after the opening brace of this function, add a statement to get a pointer to the database. Save the pointer.

```
CDatabase * pDB = m_pSet->m_pDatabase;
```

3. Use a Boolean variable to store whether the data source supports transactions. If so, begin a transaction.

```
BOOL bTrans = m_pSet->CanTransact( );
if ( bTrans  )
    pDB->BeginTrans( );
```

4. Start a **try** block. Call **Delete** for each recordset.

```
    try
    {
        m_pEmplInfoSet->Delete( );
        m_pSet->Delete( );
```

5. If the database supports transactions, commit the transaction. End the **try** block.

```
        if ( bTrans )
            pDB->CommitTrans( );
    }
```

6. In the **catch** block, inform the user of the problem, then delete the exception.

```
    catch( CDBException * pEx )
    {
        pEx->ReportError( );
        pEx->Delete( );
```

7. If a transaction is pending, call **Rollback**. End the **catch** block.

```
        if ( bTrans )
            pDB->Rollback( );
    }   //End catch
```

8. In any case, **Close** both recordsets, and then **Open** them.

```
m_pSet->Close( );   //Assume minimal ODBC support
m_pEmplInfoSet->Close( );
m_pSet->Open( );
m_pEmplInfoSet->Open( );
```

9. **Requery** the primary recordset, update the employee number in the secondary set, and **Requery** the secondary recordset. This synchronizes the data.

```
m_pSet->Requery( ); //Get a record
m_pEmplInfoSet->m_EmployeeNumberParam =
    m_pSet->m_Employee_Number;
m_pEmplInfoSet->Requery( ); //Get related data
```

10. Update the controls.

```
UpdateData( FALSE );//Show the data
```

11. Save EmployeeView.Cpp.
12. Build and run Employee.Exe.

➢ **Build and test your application**

You will be able to scroll though the records. If you change a record, and then move to a new record, the change persists. Choosing Record Clear enables you to add a record to the database. Record Delete will remove a record from the database.

The completed code for this exercise is in \Labs\C09\Lab01\Ex02.