Recordset: Dynamically Binding Data Columns (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

Recordsets manage binding table columns that you specify at design time, but there are cases when you might want to bind columns that were unknown to you at design time. This topic explains:

- When you might want to bind columns dynamically to a recordset.
- How to bind columns dynamically at run time.

> **Note**
>
> This topic applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. The techniques described generally are not recommended if you are using bulk row fetching. For more information about bulk row fetching, see **Recordset: Fetching Records in Bulk (ODBC)**.

When You Might Bind Columns Dynamically

> **Note**
>
> The MFC ODBC Consumer wizard is not available in Visual Studio 2019 and later. You can still create a consumer manually.

At design time, the MFC Application Wizard or MFC ODBC Consumer Wizard (from **Add Class**) creates recordset classes based on the known tables and columns on your data source. Databases can change between when you design them and later when your application uses those tables and columns at run time. You or another user might add or drop a table or add or drop columns from a table that your application's recordset relies on. This probably is not a concern for all data-access applications, but if it is for yours, how can you cope with changes in the database schema, other than by redesigning and

recompiling? The purpose of this topic is to answer that question.

This topic describes the most common case in which you might bind columns dynamically — having begun with a recordset based on a known database schema, you want to handle additional columns at run time. The topic further assumes that the additional columns map to `CString` field data members, the most common case, although suggestions are supplied to help you manage other data types.

With a small amount of extra code, you can:

- Determine what columns are available at run time.
- Bind additional columns to your recordset dynamically, at run time.

Your recordset still contains data members for the columns you knew about at design time. It also contains a small amount of extra code that dynamically determines whether any new columns have been added to your target table and, if so, binds these new columns to dynamically allocated storage (rather than to recordset data members).

This topic does not cover other dynamic binding cases, such as dropped tables or columns. For those cases, you need to use ODBC API calls more directly. For information, see the ODBC Programmer's Reference.

How to Bind Columns Dynamically

To bind columns dynamically, you must know (or be able to determine) the names of the additional columns. You must also allocate storage for the additional field data members, specify their names and their types, and specify the number of columns you are adding.

The following discussion mentions two different recordsets. The first is the main recordset that selects records from the target table. The second is a special column recordset used to get information about the columns in your target table.

General Process

At the most general level, you follow these steps:

1. Construct your main recordset object.

Optionally, pass a pointer to an open `CDatabase` object or be able to supply connection information to the column recordset in some other way.

2. Take steps to add columns dynamically.

   See the process described in Adding the Columns below.

3. Open your main recordset.

   The recordset selects records and uses record field exchange (RFX) to bind both the static columns (those mapped to recordset field data members) and the dynamic columns (mapped to extra storage that you allocate).

Adding the Columns

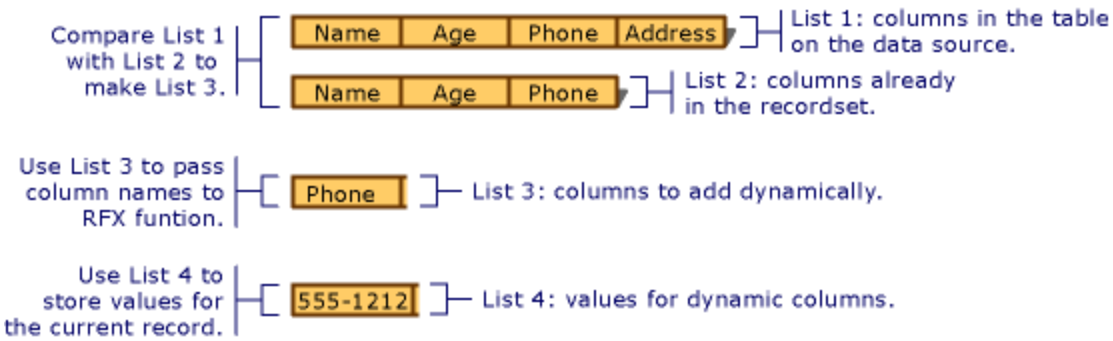
Dynamically binding added columns at run time requires the following steps:

1. Determine at run time what columns are in the target table. Extract from that information a list of the columns that have been added to the table since your recordset class was designed.

   A good approach is to use a column recordset class designed to query the data source for column information for the target table (such as column name and data type).

2. Provide storage for the new field data members. Because your main recordset class does not have field data members for unknown columns, you must provide a place to store the names, result values, and possibly data type information (if the columns are different data types).

   One approach is to build one or more dynamic lists, one for the new columns' names, another for their result values, and a third for their data types (if necessary). These lists, particularly the value list, provide the information and the necessary storage for binding. The following figure illustrates building the lists.



Building Lists of Columns to Bind Dynamically

3. Add an RFX function call in your main recordset's `DoFieldExchange` function for each added column. These RFX calls do the work of fetching a record, including the additional columns, and binding the columns to recordset data members or to your dynamically supplied storage for them.

   One approach is to add a loop to your main recordset's `DoFieldExchange` function that loops through your list of new columns, calling the appropriate RFX function for each column in the list. On each RFX call, pass a column name from the column name list and a storage location in the corresponding member of the result value list.

Lists of Columns

The four lists you need to work with are shown in the following table.

Expand table

List	Description
Current-Table-Columns	(List 1 in the illustration) A list of the columns currently in the table on the data source. This list might match the list of columns currently bound in your recordset.
Bound-Recordset-Columns	(List 2 in the illustration) A list of the columns bound in your recordset. These columns already have RFX statements in your `DoFieldExchange` function.
Columns-To-Bind-Dynamically	(List 3 in the illustration) A list of columns in the table but not in your recordset. These are the columns you want to bind dynamically.
Dynamic-Column-Values	(List 4 in the illustration) A list containing storage for the values retrieved from the columns you bind dynamically. Elements of this list correspond to those in Columns-to-Bind-Dynamically, one to one.

Building Your Lists

With a general strategy in mind, you can turn to the details. The procedures in the rest of this topic show you how to build the lists shown in Lists of Columns. The procedures guide you through:

- Determining the names of columns not in your recordset.
- Providing dynamic storage for columns newly added to the table.
- Dynamically adding RFX calls for new columns.

Determining Which Table Columns Are Not in Your Recordset

Build a list (Bound-Recordset-Columns, as in List 2 in the illustration) that contains a list of the columns already bound in your main recordset. Then build a list (Columns-to-Bind-Dynamically, derived from Current-Table-Columns and Bound-Recordset-Columns) that contains column names that are in the table on the data source but not in your main recordset.

To determine the names of columns not in the recordset (Columns-to-Bind-Dynamically)

1. Build a list (Bound-Recordset-Columns) of the columns already bound in your main recordset.

   One approach is to create Bound-Recordset-Columns at design time. You can visually examine the RFX function calls in the recordset's `DoFieldExchange` function to get these names. Then, set up your list as an array initialized with the names.

   For example, the illustration shows Bound-Recordset-Columns (List 2) with three elements. Bound-Recordset-Columns is missing the Phone column shown in Current-Table-Columns (List 1).

2. Compare Current-Table-Columns and Bound-Recordset-Columns to build a list (Columns-to-Bind-Dynamically) of the columns not already bound in your main recordset.

One approach is to loop through your list of columns in the table at run time (Current-Table-Columns) and your list of columns already bound in your recordset (Bound-Recordset-Columns) in parallel. Into Columns-to-Bind-Dynamically put any names in Current-Table-Columns that do not appear in Bound-Recordset-Columns.

For example, the illustration shows Columns-to-Bind-Dynamically (List 3) with one element: the Phone column found in Current-Table-Columns (List 1) but not in Bound-Recordset-Columns (List 2).

3. Build a list of Dynamic-Column-Values (as in List 4 in the illustration) in which to store the data values corresponding to each column name stored in your list of columns to bind dynamically (Columns-to-Bind-Dynamically).

   The elements of this list play the role of new recordset field data members. They are the storage locations to which the dynamic columns are bound. For descriptions of the lists, see Lists of Columns.

Providing Storage for the New Columns

Next, set up storage locations for the columns to be bound dynamically. The idea is to provide a list element in which to store each column's value. These storage locations parallel the recordset member variables, which store the normally bound columns.

To provide dynamic storage for new columns (Dynamic-Column-Values)

1. Build Dynamic-Column-Values, parallel to Columns-to-Bind-Dynamically, to contain the value of the data in each column.

   For example, the illustration shows Dynamic-Column-Values (List 4) with one element: a `CString` object containing the actual phone number for the current record: "555-1212".

   In the most common case, Dynamic-Column-Values has elements of type `CString`. If you are dealing with columns of varying data types, you need a list that can contain elements of a variety of types.

The result of the preceding procedures is two main lists: Columns-to-Bind-Dynamically containing the names of columns and Dynamic-Column-Values containing the values in

the columns for the current record.

> **Tip**
>
> If the new columns are not all of the same data type, you might want an extra parallel list containing items that somehow define the type of each corresponding element in the column list. (You can use the values AFX_RFX_BOOL, AFX_RFX_BYTE, and so on, for this if you want. These constants are defined in AFXDB.H.) Choose a list type based on how you represent the column data types.

Adding RFX Calls to Bind the Columns

Finally, arrange for the dynamic binding to occur by placing RFX calls for the new columns in your `DoFieldExchange` function.

To dynamically add RFX calls for new columns

1. In your main recordset's `DoFieldExchange` member function, add code that loops through your list of new columns (Columns-to-Bind-Dynamically). In each loop, extract a column name from Columns-to-Bind-Dynamically and a result value for the column from Dynamic-Column-Values. Pass these items to an RFX function call appropriate to the data type of the column. For descriptions of the lists, see Lists of Columns.

In the common case, in your `RFX_Text` function calls you extract `CString` objects from the lists, as in the following lines of code, where Columns-to-Bind-Dynamically is a `CStringList` called `m_listName` and Dynamic-Column-Values is a `CStringList` called `m_listValue`:

```
C++
```

```
RFX_Text( pFX,
            m_listName.GetNext( posName ),
            m_listValue.GetNext( posValue ));
```

For more information about RFX functions, see Macros and Globals in the *Class Library Reference*.

> **Tip**
>
> If the new columns are different data types, use a switch statement in your loop to call the appropriate RFX function for each type.

When the framework calls `DoFieldExchange` during the `Open` process to bind columns to the recordset, the RFX calls for the static columns bind those columns. Then your loop repeatedly calls RFX functions for the dynamic columns.

See also

Recordset (ODBC)
Recordset: Working with Large Data Items (ODBC)