Recordset: Parameterizing a Recordset (ODBC)

Article • 08/02/2021

This topic applies to the MFC ODBC classes.

Sometimes you might want to be able to select records at run time, using information you have calculated or obtained from your end user. Recordset parameters let you accomplish that goal.

This topic explains:

- The purpose of a parameterized recordset.
- When and why you might want to parameterize a recordset.
- How to declare parameter data members in your recordset class.
- How to pass parameter information to a recordset object at run time.

Parameterized Recordsets

A parameterized recordset lets you pass parameter information at run time. This has two valuable effects:

- It might result in better execution speed.
- It lets you build a query at run time, based on information not available to you at design time, such as information obtained from your user or calculated at run time.

When you call `Open` to run the query, the recordset uses the parameter information to complete its **SQL SELECT** statement. You can parameterize any recordset.

When to Use Parameters

Typical uses for parameters include:

- Passing run-time arguments to a predefined query.

To pass parameters to a stored procedure, you must specify a complete custom ODBC **CALL** statement — with parameter placeholders — when you call `Open`, overriding the recordset's default SQL statement. For more information, see CRecordset::Open in the *Class Library Reference* and SQL: Customizing Your Recordset's SQL Statement (ODBC) and Recordset: Declaring a Class for a Predefined Query (ODBC).

- Efficiently performing numerous requeries with different parameter information.

  For example, each time your end user looks up information for a particular student in the student registration database, you can specify the student's name or ID as a parameter obtained from the user. Then, when you call your recordset's `Requery` member function, the query selects only that student's record.

  Your recordset's filter string, stored in `m_strFilter`, might look like this:

```
C++
```

```
"StudentID = ?"
```

  Suppose you obtain the student ID in the variable `strInputID`. When you set a parameter to `strInputID` (for example, the student ID 100) the value of the variable is bound to the parameter placeholder represented by the "?" in the filter string.

  Assign the parameter value as follows:

```
C++
```

```
strInputID = "100";
...
m_strParam = strInputID;
```

  You would not want to set up a filter string this way:

```
C++
```

```
m_strFilter = "StudentID = 100";    // 100 is incorrectly quoted
                                    // for some drivers
```

For a discussion of how to use quotes correctly for filter strings, see Recordset: Filtering Records (ODBC).

The parameter value is different each time you requery the recordset for a new student ID.

> **Tip**
>
> Using a parameter is more efficient than simply a filter. For a parameterized recordset, the database must process a SQL **SELECT** statement only once. For a filtered recordset without parameters, the **SELECT** statement must be processed each time you `Requery` with a new filter value.

For more information about filters, see Recordset: Filtering Records (ODBC).

Parameterizing Your Recordset Class

> **Note**
>
> This section applies to objects derived from `CRecordset` in which bulk row fetching has not been implemented. If you are using bulk row fetching, implementing parameters is a similar process. For more information, see **Recordset: Fetching Records in Bulk (ODBC)**.

Before you create your recordset class, determine what parameters you need, what their data types are, and how the recordset uses them.

To parameterize a recordset class

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

1. Run the MFC ODBC Consumer Wizard from **Add Class** to create the class.

2. Specify field data members for the recordset's columns.

3. After the wizard writes the class to a file in your project, go to the .h file and manually add one or more parameter data members to the class declaration. The addition might look something like the following example, part of a snapshot class designed to answer the query "Which students are in the senior class?"

```
C++

class CStudentSet : public CRecordset
{
// Field/Param Data
    CString m_strFirstName;
    CString m_strLastName;
    CString m_strStudentID;
    CString m_strGradYear;

    CString m_strGradYrParam;
};
```

   Add your parameter data members after the wizard-generated field data members. The convention is to append the word "Param" to each user-defined parameter name.

4. Modify the DoFieldExchange member function definition in the .cpp file. Add an RFX function call for each parameter data member you added to the class. For information about writing your RFX functions, see Record Field Exchange: How RFX Works. Precede the RFX calls for the parameters with a single call to:

```
C++

pFX->SetFieldType( CFieldExchange::param );
// RFX calls for parameter data members
```

5. In the constructor of your recordset class, increment the count of parameters, `m_nParams`.

   For information, see Record Field Exchange: Working with the Wizard Code.

6. When you write the code that creates a recordset object of this class, place a "?" (question mark) symbol in each place in your SQL statement strings where a parameter is to be replaced.

At run time, "?" placeholders are filled, in order, by the parameter values you pass. The first parameter data member set after the SetFieldType call replaces the first "?" in the SQL string, the second parameter data member replaces the second "?", and so on.

> **Note**
>
> Parameter order is important: the order of RFX calls for parameters in your `DoFieldExchange` function must match the order of the parameter placeholders in your SQL string.

> **Tip**
>
> The most likely string to work with is the string you specify (if any) for the class's **m_strFilter** data member, but some ODBC drivers might allow parameters in other SQL clauses.

Passing Parameter Values at Run Time

You must specify parameter values before you call `Open` (for a new recordset object) or `Requery` (for an existing one).

To pass parameter values to a recordset object at run time

1. Construct the recordset object.
2. Prepare a string or strings, such as the `m_strFilter` string, containing the SQL statement, or parts of it. Put "?" placeholders where the parameter information is to go.
3. Assign a run-time parameter value to each parameter data member of the object.
4. Call the `Open` member function (or `Requery`, for an existing recordset).

For example, suppose you want to specify a filter string for your recordset using information obtained at run time. Assume you have constructed a recordset of class `CStudentSet` earlier — called `rsStudents` — and now want to requery it for a particular

kind of student information.

C++

```
// Set up a filter string with
// parameter placeholders
rsStudents.m_strFilter = "GradYear <= ?";

// Obtain or calculate parameter values
// to pass--simply assigned here
CString strGradYear = GetCurrentAcademicYear( );

// Assign the values to parameter data members
rsStudents.m_strGradYrParam = strGradYear;

// Run the query
if( !rsStudents.Requery( ) )
    return FALSE;
```

The recordset contains records for those students whose records meet the conditions specified by the filter, which was constructed from run-time parameters. In this case, the recordset contains records for all senior students.

> **Note**
>
> If needed, you can set the value of a parameter data member to Null, using **SetParamNull**. You can likewise check whether a parameter data member is Null, using **IsFieldNull**.

See also

Recordset (ODBC)
Recordset: Adding, Updating, and Deleting Records (ODBC)
Recordset: How Recordsets Select Records (ODBC)