Oracle Objects for OLE C++ Class Library Workbook

INTRODUCTION

This Workbook provides examples of C++ code that use the Oracle Objects for OLE C++ Class Library. The *Oracle Objects for OLE C++ online help system* provides general information on Oracle Objects for OLE as well as detailed descriptions of all the C++ classes and their methods.

Some of the examples discussed in this Workbook are so simple that they are represented only with code fragments. Others are sufficiently complex that the complete code has been provided in separate files. The notes for those examples indicate the names of the files containing the code. The sample files are found in subdirectories of the *samples* directory, which is a subdirectory of the *workbook* directory. The samples are installed by default or if you choose to install sample code in a custom install. A few of the examples provide complete projects.

The examples presented here begin with simple, fairly common uses of the library and progress to more complex, less common uses. Clever and dedicated developers (with time on their hands) may learn how to use the Class Library simply by working through the examples. However, using the library will be easier if you read and understand relevant sections of the *Oracle Objects for OLE C++ online help system*.

This Class Library is intended for use against an Oracle database—a relational database that uses the SQL language as its primary interface. The SQL used in these examples is as simple as possible, and brief explanations are provided of what the SQL statement does in each case. But to exploit the full power of this Library—and of any relational database—you need a good grasp of SQL.

Many of the examples use the standard demonstration tables that are shipped with Oracle databases. Most of the examples entail a database named “ExampleDB”, a user named “scott”, and a password for that user of “tiger”. To use the example code for your own projects, you need to change the database name to refer to your database and the user information to refer to some valid user on your database. (Alternatively, you can create a database alias called ExampleDB that refers to a database with the scott/tiger account.)

To use the Class Library properly, your application must call the **OStartup** and **OShutdown** methods. These routines perform necessary initialization and cleanup for the Class Library (for example, they initialize and uninitialize OLE). In the examples that contain only fragments of code, we assume that these routines are called outside the fragments.

<u>Example 1: Just get some data</u>

This very minimal example demonstrates the simplest and most common use of the library: to fetch some simple data from the database. In this example look at a table named “emp”, which has several columns. The only column we care about for this example is one called “sal”, which represents the salary paid to an employee. The database contains a record for each current employee.

Our task is to find determine the total salary paid—the sum of the salaries of all the employees. To do this we have to

1. connect to the database,
2. query the database and retrieve records, and
3. process the records.

We can do this all in a single subroutine, as follows:

```
double SumSalary(void)
{
   ODatabase datab;   // the database object
   ODynaset  dyn;     // dynaset object
   double    sum=0.0; // sum of all salaries
   double    cursal;  // salary of the current employee

   // connect to the database
   datab.Open("ExampleDB", "scott", "tiger");

   // query the database
   dyn.Open(datab, "select sal from emp");

   // process all the records
   while (!dyn.IsEOF()) // until we’ve gone past all the records
   {
       dyn.GetFieldValue("sal", &cursal);  // get the current salary
       sum += cursal;
       dyn.MoveNext();    // move to the next record
   }

   return(sum);
}
```

For the sake of simplicity, we have ignored all error handling in this example. In later examples we consider how to handle various errors.

Both the database and the dynaset objects are “opened". This is necessary because an unopened **ODatabase** has no connection to a database, and an unopened **ODynaset** has no records. Until these objects are opened they are not very useful. The database connection is established by opening the **ODatabase** object by passing in the name of the database (ExampleDB), the username (scott), and the user’s password (tiger). The **ODynaset** is opened on a particular **ODatabase**. **ODynasets** are always the set of records that is the result of an SQL select on the database. The database is represented by the **ODatabase** argument to the **ODynaset Open** method. The **ODynaset Open** method is also given a SQL statement that indicates which records from the database are to be fetched.

In this case the SQL statement is “select sal from emp” (“emp” is the name of the table being queried). A relational database contains many tables. A query can access one or more of the tables at once. A query against an Oracle database can, in fact, access tables across many databases distributed on a network. The emp table has a number of columns in it; “sal”, “ename”, and “hiredate” are three of them. Since we are only interested in salary, we ask only for the “sal” column. We could have asked for more columns (the special symbol “*” in the column list gets all the columns). This would not have changed our code, but it would have fetched more data from the database—unnecessarily.

Opening the **ODynaset** gives us a set of records that match the data in the database. The **ODynaset** contains the notion of which record is "current". We can use the navigational methods on the **ODynaset** to move from record to record. By default, when the **ODynaset** is opened, the first record becomes current. We can then use the **ODynaset**’s **MoveNext** method to traverse all the records that were fetched.

We use the **IsEOF** method to tell when we’ve gone through all the records. It returns TRUE when we’ve "Move[d]Next" past the last record.

Now that we can navigate through the records, it's time to fetch the data. The simplest way to do this is to use the **GetFieldValue** method of the **ODynaset**. This method is overloaded to support different types. The method converts the data from the database to the type you implicitly ask for, if possible. Here we ask for a double, because the salary is stored as a value with fractional dollars. We call **GetFieldValue** with the name of the column and pass the address of a double variable we want set to the value of the salary in the current record.

Then we add up all the salaries. Sounds simple? Actually, it is, because the server can do most of the work for us. There’s no need for us to download all the records (in a big company there may be thousands, and that’s a lot of network traffic). Instead, we can let the server do the sum for us.

We replace the line

```
   dyn.Open(datab, "select sal from emp");
```

with

```
   dyn.Open(datab, "select sum(sal) from emp");
```

Now the database hands us back a single record that contains the sum of the salaries. Often (as in this case), the server is a more powerful computer than our client workstation. In such cases, it is more efficient to let the server do the simple calculations, and it decreases the total number of bytes that have to be transferred over the network.

If we use this statement as it stands, subsequent references to the returned column need to use the name “sum(sal)”, which is clumsy because of the parentheses. One more modification to the SQL statement can make it more elegant. We say:

```
   dyn.Open(datab, "select sum(sal) sumsal from emp");
```

and now we can refer to the returned column as “sumsal”. This renaming can become significant in the case of more complex calculations or when we later change queries but want to keep the same column names.

Our completed routine becomes:

```
double SumSalary(void)
{
   ODatabase datab;   // the database object
   ODynaset  dyn;     // dynaset object
   double    sum;     // sum of all salaries

   // connect to the database
   datab.Open("ExampleDB", "scott", "tiger");

   // query the database
   dyn.Open(datab, "select sum(sal) ""sumsal"" from emp");

   // get the sum of the salaries
   dyn.GetFieldValue("sumsal", &sum);  // get the salary total

   return(sum);
}
```

Notice one more thing. We connected to the database but we never disconnected. This is correct. When the routine exits, it destroys the **ODynaset** and **ODatabase** objects. The destruction of the **ODatabase** object properly drops the database connection—you don’t have to think about it.

Example 2: Execute a SQL statement

Another typical operation with a database is to execute some SQL statement that is not a query. For example, you execute SQL statements to create tables, add users, administer the database, delete a set of records, and so forth. In this example we create a simple table. This example also demonstrates some simple error handling.

```
// routine to create the states table.
//   returns 0 on success, -1 on failure

// There is a bug in this code!  (See below)  Don't use this!

int CreateStatesTable(void)
{
   ODatabase datab;   // the database object
   oresult   ores;    // indicates whether operation succeeded

   // connect to the database
   ores = datab.Open("ExampleDB", "scott", "tigers");
   if (ores != O_SUCCESS)
   { // couldn't open the database connection
       ErrorMessage(datab.GetSession().GetServerErrorText());
       return(-1);
   }

   // create the table
   const char *sqls = "create table states (name char(15), area number,
population number)";

   ores = datab.ExecuteSQL(sqls);
   if (ores != O_SUCCESS)
   {   // couldn't create the table
       ErrorMessage(datab.GetSession().GetServerErrorText());
       return(-1);
   }

   // everything went just fine
   return(0);
}
```

We connect to the database by opening the **ODatabase** object. Most methods in the library return a result of the type oresult, which indicates whether the method succeeded or failed. If you determine that the method failed, you need to call other routines to get the actual error. In this case we want the routine **GetServerErrorText**, which is a method in the **OSession** class, to get the error text back from the Oracle database software.

We check the return from the **Open** method. If the routine did not execute successfully, we get an **OSession** object from the **ODatabase** object and use it to get the error text. Notice that since the **GetSession** method returns an **OSession**, it can be used inline. Since the C++ objects in the Class Library are handles to underlying implementation objects, they are lightweight. The use of temporary objects, such as the **OSession** that is returned by **GetSession**, is perfectly reasonable. **GetServerErrorText** returns a (const char *), which we hand to some generic error processing routine, which presumably alerts the user.

To create the table, we first construct a SQL statement. In this example we are creating a trivial table, so we can use a static string. In general, code puts together some complex SQL statement. Our SQL statement creates a table with the name “states” that has three columns in it: “name”, which is a text column with a width of 15 characters, “area”, which is a number, and “population”, which is another number.

Next we use the **ExecuteSQL** method to pass the SQL statement to the database to be executed. Once again we check the return and get the error text if necessary.

Because the cleanup work is done for you in the object destructors, you don’t have to worry about the database connection—whether or not the open succeeded. Cleanup happens automatically.

Now, when we run this example it fails (we’ve given the wrong password for the user scott). And **GetServerErrorText** returns a 0 rather than an error string indicating a bad password. Why? Because **GetSession** is returning an unopened **OSession** object. And that’s because our database object can’t return an **OSession** because it isn’t open. But we’re getting the error on the Open! How do we get the correct error?

The answer is first to create an **OSession** object explicitly, and then open the database using that session. Then, if the database open fails, we can refer to our **OSession** object. Rewritten, the code looks like this:

```
// routine to create the states table.
//   returns 0 on success, -1 on failure

// Corrected version

int CreateStatesTable(void)
{
   OSession  sess;    // database session object
   ODatabase datab;   // the database object
   oresult   ores;    // indicates whether operation succeeded

   // open the default (unnamed) session
   ores = sess.Open();
   if (ores != O_SUCCESS)
   {
       ErrorMessage(sess.GetErrorText());
       return(-1);
   }

   // connect to the database
   ores = datab.Open(sess, "ExampleDB", "scott", "tigers");
   if (ores != O_SUCCESS)
   { // couldn't open the database connection
       ErrorMessage(sess.GetServerErrorText());
       return(-1);
   }

   // create the table
   const char *sqls = "create table states (name char(15), area number,
population number) ";

   ores = datab.ExecuteSQL(sqls);
   if (ores != O_SUCCESS)
   {   // couldn't create the table
       ErrorMessage(sess.GetServerErrorText());
       return(-1);
   }

   // everything went just fine
   return(0);
}
```

This version is almost the same as the earlier one, except that we use our explicit **OSession** object to get the database errors, and there's code to open the **OSession** object. Opening the **OSession** is very straightforward. Notice that if the **OSession Open** fails we get an error message by calling **GetErrorText** rather than **GetServerErrorText**. **GetServerErrorText** is appropriate when the error is a problem with the database. This can't be the case until we're actually connected to, or trying to connect to, the database. If the **OSession** open fails, it is due to some internal problem such as low memory. So we use **GetErrorText** instead.

Example 3: A joined query

One of the most important and powerful features of a relational database is the possibility of querying data from several tables at once. Instead of formulating and coordinating several related queries on different tables, the queries can be combined. Most of this example is about writing the SQL needed for a joined query. If you are already familiar with SQL, you can skim through this example, but be sure to read the very last paragraph.

Consider the emp and dept sample tables that are installed in the scott account of most Oracle7 sample databases. The emp table contains the employee name (ename field), the employee's job (job field), the employee's salary (sal field), and the department number where the employee works (deptno field). A related table is the dept table. It also contains the department number (deptno field), as well as the department name (dname field) and the department location (loc field).

Suppose we want to write an application that looks at where the different employees work. We want to query employee names and the location of the departments where they work. One possible solution (though not a good one) is to create two separate queries:

```
// bad solution to employee & department join
ODatabase datab;
datab.Open("ExampleDB", "scott", "tiger");   // open the database

ODynaset empdyn;  // employee query
ODynaset deptdyn; // department query
empdyn.Open(datab, "select ename from emp");
deptdyn.Open(datab, "select loc from dept");
```

Now we have one query for employee names and another for department locations. Not only do we need to manage the two queries, but we do not know which record in the department list corresponds to which employee. The two queries are not sorted: the first department is not necessarily the department of the first employee.

However, the two tables do share a field: deptno. This field is the "key" that relates the two tables. Now we could write:

```
// still not a good solution to employee & department join
ODatabase datab;
datab.Open("ExampleDB", "scott", "tiger");   // open the database

ODynaset empdyn;  // employee query
ODynaset deptdyn; // department query
empdyn.Open(datab, "select ename, deptno from emp");
deptdyn.Open(datab, "select loc, deptno from dept");
```

Now whenever we look at an employee record, we can read its department number. Then we can scan through the department records to find the department with that department number. Then we'll have the location. The trouble with this solution is that it requires us to scan those department records pretty often. When we're writing a client-server application, this is the kind of processing that the server does best. The server is very good at coordinating data; there's no reason to download the data to the client workstation for coordination.

What we can do here is tell the server to match the deptno fields of the two tables. Then we have:

```
// the joined query solution
ODatabase datab;
datab.Open("ExampleDB", "scott", "tiger");  // open the database

ODynaset jdyn;  // joined query
jdyn.Open(datab, "select emp.ename, dept.loc from emp, dept \
                 where emp.deptno = dept.deptno");
```

Now we have a dynaset with two fields in it: ename and loc. These are just the fields we wanted. And the loc field corresponds to the location of the department where the employee works. In effect, when the server was preparing the records for return to our client, it did the coordination, more efficiently than we could have done it ourselves (because of all the machinery of a relational database).

A couple of points are noteworthy:

- The first is the syntax of the SQL statement. We specified which fields we wanted in our query. Since the fields are coming from several tables, we fully specified the fields by calling them by names like “table.field”. When we specified the tables we were querying, we gave a comma-separated list of all the tables we wanted. Finally, we gave a “join condition”, which is just a "where" clause that specifies how we want to coordinate the data from the various tables.

- Another point is that the deptno field is not one of the fields in our dynaset. If we wanted deptno we would have to include it in the list of select fields as: "select emp.ename, emp.deptno...”.

- By the way, the backslash (“\”) is not part of the SQL statement. It is just the standard C syntax for continuing a literal across multiple lines.

- Finally a most important point: Dynasets formed from joined queries are essentially read-only. The “records” in such a dynaset do not correspond to actual records in any table. Therefore it is not possible to delete, update, or add new records to such a dynaset: there’s no table to be changed so it doesn’t make sense. Other kinds of queries, such as queries that have computed columns, are also read-only.

Example 4: A connection dialog

You need to specify three things to open a database object: the database name, the user name, and the user's password. You can hard code these values into your code, but that is a bad idea because then you won't be able to use the same program for another user or against a different database. It is much better to generalize the connection information.

The obvious way to do this is to create a dialog that obtains the information. We want a routine that hands back an open **ODatabase** object if the connection was possible and a closed **ODatabase** to indicate failure. We want the ability to create the ODatabase with different options.

The sample subdirectory *logdlg* contains the files LOGDLG.CPP and LOGDLG.H. These files contain the code for an implementation of such a login dialog using Visual C++'s MFC framework. The dependency on the framework is slight, so you don't have to be very experienced with MFC to understand what the code is doing.

The important portion of the logdlg class follows:

```
class logdlg : public CDialog // subclass of a dialog
{
public:
   // Construction
   logdlg(CWnd* pParent = NULL);    // standard constructor

   // get a database login
   ODatabase GetLogin(long options = ODATABASE_DEFAULT);

private:
   OSession  m_session;      // our handle to the default session
   ODatabase m_database;     // our handle to the database object

// more implementation details...
};
```

An instance of the class is a dialog, which is what is created here. Then, when the client of the class wants a valid and open **ODatabase** object, the method **GetLogin** is called. This method runs the dialog, tries to make the database connection, notifies the user of errors, and finally returns.

You can see from the source of **GetLogin** that it does very little. It performs some setup and then calls the "DoModal" routine to run the dialog. It then returns an **ODatabase** object, which may or may not be open. The real work is done when "OK" is pressed. Here's the method that gets called then:

```
void logdlg::OnOK()
{
      CString dbname;   // database name string
      CString user;     // user name string
      CString password; // password string

      // get the strings the user has entered
      GetDlgItem(IDC_USERNAME)->GetWindowText(user);
      GetDlgItem(IDC_PASSWORD)->GetWindowText(password);
      GetDlgItem(IDC_DATABASE)->GetWindowText(dbname);
      // try to open the database
      if (m_database.Open(m_session, dbname, user, password,
m_options) != OSUCCESS)
      { // some error
          // get the oracle error number
          long oraerr = m_session.ServerErrorNumber();

          // get the oracle error message, to display to the user
          const char *dberrs = m_session.GetServerErrorText();
          ErrorMessage(dberrs); // tell user what went wrong
      }
      else
      {
          // we're done - get out of the dialog
          CDialog::OnOK();
      }

      return;
}
```

This method gets the text from the login dialog and uses it to try to open the **ODatabase** object. When it is opening the **ODatabase** object it uses the options that were passed in to **GetLogin**. If there is an error, the generic "ErrorMessage" routine is called (which may just put up a simple message box). Only if the open was successful is the parent dialog class's routine "OnOK" called (which closes the dialog).

There are two ways for the dialog to end: Either the OK is successful, in which case the returned **ODatabase** object will be open, or the user chooses cancel, in which case the returned **ODatabase** object will be closed. If the user chooses "OK" to try to connect and the connection fails, the above routine alerts the user but does not close the dialog.

Note again that it is reasonable to pass **ODatabase** instances around. Here we are using one as the return value from a subroutine. A caller might have some code like this:

```
// connect to the database
ODatabase  database;
logdlg   dblogin;

database = dblogin.GetLogin();
if (database.IsOpen())
   ; // success
else
   ; // user canceled for some reason
```

Useful improvements to the dialog would be to allow the fields to be preset, or to allow database options to be set on the **ODatabase.Open**.

<u>Example 5: Updating a dynaset</u>

Sometimes you may need to do more than just read the contents of a dynaset. You may also want to modify the records in the dynaset, and therefore the database itself.

You can update database records using the Class Library in two ways. The first is to execute an update SQL statement. The second is to use the dynaset to edit the records.

Consider the emp table. It contains employee salary information. Suppose the company has had a good year and everyone is getting a $1000 salary increase.

```
// give all employees a raise of amount "raise"
oboolean GiveRaise(int raise)
{
   ODatabase db;  // our database
   logdlg    db_login;  // used to connect to database (see example 4)

   // open a connection to the database
   db = db_login.GetLogin();
   if (!db.IsOpen())
       return(FALSE);  // indicates that we couldn’t give out the raises

   // give everybody a raise
   char sql[80];  // for constructing the sql statement
   sprintf(sql, "update emp sal = sal + %d", raise);

   if (db.ExecuteSQL(sql) != OSUCCESS)
       return(FALSE);  // we couldn’t do it
                       // would be a good idea to look at errors to see
why

   return(TRUE);  // we’ve given the raises
}
```

Notice that we run a login dialog in this routine and then, by exiting the routine and destroying the **ODatabase** instance, we drop the database connection. Generally a routine like this should probably pass in the **ODatabase** object as a parameter so that many such operations can share a login.

Updating records in this way is very efficient because the server does all the work. But in more complex situations it is not possible to use such a simple update SQL statement. A more realistic example would be a case in which the raise for each employee is calculated in some fashion and then applied to each record. This requires us to go through every record in the database and update some of them as needed. Here’s a routine to do that:

```
// Calculate and apply raises to all employees
oboolean GiveRaises(const ODatabase &db)
{
   if (!db.IsOpen())
       return(FALSE);  // we can’t work without an open database

   // create a dynaset referring to the employees
   ODynaset dyn;
   dyn.Open(db, "select ename, sal, hiredate from emp");
   if (!dyn.IsOpen())
       return(FALSE);  // can’t open dynaset
   double   salary, raise;

   // go through all the records
   while (!dyn.IsEOF())
   {  // for every record

       raise = CalculateRaise(dyn);  // calculate raise somehow

       if (raise > 0)
       { // edit this record
           dyn.StartEdit();  // begin editing this record

           dyn.GetFieldValue("sal", &salary);  // get old salary
           dyn.SetFieldValue("sal", salary + raise);  // set new salary

           dyn.Update();  // finish editing this record
       }
       dyn.MoveNext();  // go to the next record
   }

   return(TRUE);
}
```

There are three steps to editing a record:

1. First, call the **StartEdit** method to tell the dynaset that you're about to edit the record.

2. Second, make calls to change values in the current record. These could be **Dynaset.SetFieldValue** calls or they may be **OField.SetValue** calls.

3. Finally, call **Update** to push the changes into the database.

This method is much more flexible than a direct update, but it is also slower. **StartEdit** must go to the database and attempt to lock the record. It also checks to see if the data in the database is different from the data in the dynaset (which would indicate that someone else has edited the record, invalidating the dynaset). If another dynaset or program has a lock on the record, **StartEdit** can either wait indefinitely or can return a "can't obtain lock" error. (See the documentation on **ODatabase** options for the ODATABASE_NOWAIT option.)

Finally there is a subtle difficulty when updating (or making any other kind of change) to dynasets or to the database: Changes made to records in a dynaset are immediately reflected in that dynaset. Any changes made by other users, or other dynasets in your program, are not reflected immediately. If the dynaset has not yet fetched the row that was changed, it gets the new value when and if it does fetch the row. But it is difficult to predict which rows have really been fetched (because of caching). You can guarantee that the dynaset has the most up-to-date version of the data by performing a **Refresh** on the dynaset.

Example 6: Adding and deleting records with a dynaset

Example 5 covers the editing of existing records. In many cases you need not only to edit records but to add new records and to delete existing ones. The **ODynaset** method **DeleteRecord** deletes the current record. To delete a record, navigate to it and then execute **DeleteRecord**.

Here is a routine that deletes all employees with a salary above a certain amount.

```
oboolean DeleteOverpaid(const ODatabase &db, double maxsal)
{
   if (!db.IsOpen())
       return(FALSE);  // error because database object isn't open

   ODynaset dyn;
   OField   salary;

   dyn.Open(db, "select sal from emp");
   salary = dyn.GetField("sal");
   while (!dyn.IsEOF())
   {
       if ((double) salary > maxsal)
       { // this employee is overpaid.  Delete.
           dyn.DeleteRecord();  // deletes the current record
       }
       dyn.MoveNext();  // go through all the records
   }

   return(TRUE);
}
```

Once we delete a record with **DeleteRecord**, the current record becomes invalid (it has been deleted). Most operations on such a record fail. Navigating to another record gets us to a valid record. We won't be able to navigate back to the deleted (thus invalid) record.

In this example we get the field value by using an **OField** object rather than directly through the **ODynaset** object. **OField** objects are closely linked to the underlying dynaset: as navigation occurs in the dynaset, the field value changes to contain the current record value. An **OField** instance is more convenient to use than going through the dynaset, because it can be treated as a variable, as illustrated by the "(double) salary > maxsal" statement above.

Adding a new record is a little more complex. You can use two methods to add records to a dynaset. The first is **AddNewRecord**, which adds a blank record. The second is **DuplicateRecord**, which adds a blank record and then fills it with the field values found in the current record (current at the time **DuplicateRecord** is called). When a blank record is created, it is either filled with NULLs or sent to the database server to have defaults filled in, depending on whether the ODATABASE_PARTIAL_INSERT option has been turned on. (See the **ODatabase** documentation for details.)

After adding the record you may change some of its values. The **AddNewRecord** or **DuplicateRecord** call allows editing similarly to the way that **StartEdit** does. When you have made all the changes you want, call **Update** to put your changes into the database.

When you are adding new records, it is important to be aware of the structure of the table that you are editing. Some fields may require unique values. Some fields may not allow NULL values. These requirements must be fulfilled or the **Update** call fails.

Here is a routine that adds a new record to the employee table.

```
oboolean AddEmployee(ODynaset *empdyn, const char *ename,
                    double salary, int deptno)
{
   if (!empdyn->IsOpen())
       return(FALSE);  // can't work with unopened dynaset

   // Add the new record
   if (empdyn->AddNewRecord() != OSUCCESS)
       return(FALSE);

   // Set the values of the record
   empdyn->SetFieldValue("ename", ename);
   empdyn->SetFieldValue("sal", salary);
   empdyn->SetFieldValue("deptno", deptno);

   // the empno field is required and must be unique.
   //  We will find the current maximum empno and add 1 to it.

   ODynaset empnodyn;
   empnodyn.Open(empdyn->GetDatabase(),
                "select max(empno)+1 newempno from emp");
   int empno;
   empnodyn.GetFieldValue("newempno", &empno);

   // finish setting the employee record
   empdyn->SetFieldValue("empno", empno);

   // put this record in the database
   if (empdyn->Update() != OSUCCESS)
       return(FALSE);

   return(TRUE);
}
```

The values of most of the fields were left NULL. We set the values of ename, sal, and deptno based on the arguments to the routine.

The empno field is used as a unique key in the employee table, so it must be unique and non-NULL. An easy way to specify this is to ask the server for the highest current employee number, and then have the server add 1 to that number so that we get a new unique value.

In practice this is not a good solution, for two reasons. First, we are making an extra query to the database for every record added. It would be faster to figure out a good employee number at the beginning of the program and then use that number (incrementing it or whatever). Second, this method guarantees only that the empno field is unique among current records. If an employee record was deleted, its employee number field might get reused. This is generally not desirable. The database has objects called “sequences” that help with this kind of problem.

As with editing and updating a dynaset, there is a difficulty with cross-dynaset consistency. If a dynaset contains a record, and it has been fetched from the server already, then when another dynaset deletes that same record there will be a “ghost” record in the first dynaset. Similarly, records added to the first dynaset won’t show up in other existing dynasets. (Added records can be seen by refreshing the dynaset.) Attempting to edit a record that has been deleted from the database will fail on the **StartEdit** method.

Example 7: Transaction control

Normally, when you change the database data with dynaset operations, the database immediately reflects the changes you have made. When you add records or change field values, the database is changed when you execute the **Update** method. When you delete records, the database is changed when the **DeleteRecord** method is executed. This is sufficient when changing a record is an independent operation.

But sometimes you have an entire set of changes that must be made together. Either they are all made or none of them are made. The classic example is an application that maintains bank account balances. When a transfer is made from one account to the other, two separate records are edited. The record that records the balance of one account is debited and the record of the other account is credited. These two operations must succeed or fail together. If one takes place without the other, the bank's books will not balance properly.

The solution is to enclose the set of changes in a transaction. Before making the first change you call the **BeginTransaction** method. Then you make your changes. If you want to cancel the changes, you call the **Rollback** method. If you want to make the changes permanent, you call the **Commit** method. Either **Rollback** or **Commit** completes the current transaction. Then, by default, we return to the normal behavior of changing the database directly on each **Update** or **DeleteRecord**.

Note to experienced Oracle developers: The normal transactional model of the Class Library is similar to the autocommit mode in SQL*Plus or Oracle Forms. Executing **BeginTransaction** is like turning autocommit off. Executing **Rollback** or **Commit** is a rollback or commit. And the default behavior of **Rollback** or **Commit** is then to reenter autocommit mode.

The example we consider here uses **DynasetMarks**. These are objects that remember a position in a dynaset. You can get a mark on the current position and then later use the **MoveToMark** method to reposition on that record again.

Here's a bank account example:

```
// transfer money from credit to debit account
// we have a DynasetMark on the two accounts
void TransferMoney(ODynaset accounts,   // dynaset of bank accounts
                   ODynasetMark credit,  // mark on record to be credited
                   ODynasetMark debit,   // mark on record to be debited
                   double amount         // amount to transfer
                   )
{
   // get the dynaset's session
   OSession banksess = accounts.GetSession();

   // start the transaction
   banksess.BeginTransaction();

   // make the transfer
   double balance;  // an account's balance

   // credit one account
   accounts.MoveToMark(credit);
   accounts.StartEdit();
   accounts.GetFieldValue("balance", &balance);
   accounts.SetFieldValue("balance", balance + amount);
   if (accounts.Update() != OSUCCESS)
   { // couldn't change the first record
       banksess.Rollback();
       return;
   }

   // debit the other account
   accounts.MoveToMark(debit);
   accounts.StartEdit();
   accounts.GetFieldValue("balance", &balance);
   accounts.SetFieldValue("balance", balance - amount);
   if (accounts.Update() != OSUCCESS)
   { // couldn't change this record
       banksess.Rollback();
       return;
   }

   // everything is fine - commit the transaction
   banksess.Commit();

   return;
}
```

(Here we're checking for errors only on the **Update** statement. A real application would be more careful and would check for errors on all the methods.)

How could the second update fail and the first succeed? The database connection might have been lost (maybe because of a hardware failure in a network). Or perhaps the new account balance violated a database constraint (for example, there might be a trigger stored in the database that caused the update to fail because the account balance dropped below zero).

Example 8 shows another way to accomplish the same task. Rather than using a dynaset, it calls **ExecuteSQL** directly to make the database changes. Changes made to the database using **ExecuteSQL** (by way of update, insert, and delete statements) are part of the transaction that is managed by **BeginTransaction** and **Commit** or **Rollback**.

The **Commit** and **Rollback** methods take an oboolean argument named *startnew*, which is FALSE by default. If you set it to TRUE, a new transaction is started immediately after the **Commit** or **Rollback** (just as if you had called **BeginTransaction** again).

Oracle developers note: If you call **BeginTransaction** immediately after opening the session and always set *startnew* to TRUE, you will have a transactional environment similar to the Oracle environment to which you are accustomed.) Consider the following code:

```
// fragment illustrating Oracle7 transaction details

OSession sess;
sess.Open();  // open the default session
sess.BeginTransaction();  // start a transaction
ODatabase db;
db.Open(sess, "ExampleDB", "scott", "tiger");
ODynaset dyn1;
dyn1.Open(db, "select * from emp order by empno");
```

```
// change the first record
dyn1.MoveFirst();
dyn1.StartEdit();
dyn1.SetFieldValue("sal", 7500);
dyn1.Update();

// open another dynaset
ODynaset dyn2;
dyn2.Open(db, "select * from emp order by empno");
dyn2.MoveFirst();
long salary2;
dyn2.GetFieldValue("sal", &salary2);

// open yet another dynaset
// open a named session
OSession nameds;
nameds.Open(sess.GetClient(), "session2");
ODatabase db2;
db2.Open(nameds, "ExampleDB", "scott", "tiger");
ODynaset dyn3;
dyn3.Open(db2, "select * from emp order by empno");
dyn3.MoveFirst();
long salary3;
dyn3.GetFieldValue("sal", &salary3);
```

The second **ODynaset** was opened within the same session as the first. The first **ODynaset** made a change that has not yet been committed to the database. The third **ODynaset** was opened on a different session. What are the values of salary2 and salary3?

Because dyn2 is in the same session as dyn1, it sees the same database state. It will see a salary of 7500. But dyn3 is in a different session, so it will not see a salary of 7500; rather, it will see the salary that was in the field before it was changed. This result is independent of whether dyn3 is a part of the same process or not. Dyn3 could be on another computer altogether.

<u>Example 8: Parameters</u>

All of the SQL statements we have used in the previous examples have been entirely literal. So, when we wanted the employee records of Department 20, we use the SQL statement: "select * from emp where deptno = 20". And if we then wanted the records from Department 10, we used "select * from emp where deptno = 10". This is inefficient in a number of ways. It would be more efficient if we could say: "select * from emp where deptno = :deptno", where *:deptno* is something that can be set to a value at a later time. This is precisely what a parameter is: A parameter allows you to introduce variables into the processing of an SQL statement.

Parameters are attached to an **ODatabase** object and are accessed by way of an object called **OParameterCollection**. The **OParameterCollection** object exposes a variable number of parameters, and you can add or remove parameters from the collection. (The other collection types—**OFieldCollection**, **OConnectionCollection**, and **OSessionCollection**—are read-only.) You can create as many parameters as you want. At any given time, any parameters that are auto-enabled will be bound to your SQL statement. (For more information, consult the *Oracle Objects for OLE C++ online help system*.

The following code uses a parameter to select different sets of employee records.

```
// main routine (processes departments)
ProcessCompany(ODatabase *db)
{
   // get a list of the department numbers
   ODynaset deptnumbers;
   deptnumbers.Open(*db, "select deptno from dept order by deptno");
   OField deptnof = deptnumbers.GetField("deptno"); // the deptno column

   // set up a "dnumber" parameter on the database
   OParameterCollection params = db->GetParameters();
   // create parameter with initial value of 0
   params.Add("dnumber", 0, OPARAMETER_INVAR, OTYPE_NUMBER);
   OParameter dnumber = params.GetParameter("dnumber");

   // process every department
   deptnumbers.MoveFirst();
   while (!deptnumbers.IsEOF())
   {
       // set the value of the parameter
       dnumber.SetValue((int) deptnof);

       // process that department
       ProcessDepartment(db);

       // go to next department
       deptnumbers.MoveNext();
   }

   // all done.  We don't need the dnumber parameter to be part of
   // the collection anymore so get rid of it
   params.Remove("dnumber");

   return;
}

void ProcessDepartment(ODatabase *db)
{
   // open a dynaset of employees for the current department
   ODynaset emps;
   emps.Open(*db, "select * from emp where deptno = :dnumber");

   // process them all
   emps.MoveFirst();
   while (!emps.IsEOF())
   {
       // do something interesting here
       emps.MoveNext();
   }

   return;
}
```

In ProcessCompany we set up the parameter that is then used by the subroutine ProcessDepartment. We can then write the ProcessDepartment routine more generally. The department number it is processing is not hard-coded into it anywhere; rather, it depends on the parameter "dnumber" existing for the database, which is its argument.

The parameter is created by using the **Add** method on the **ODatabase**'s parameter collection. The parameter's value can be changed at any later time. However, changing the parameter's value does not instantly change the contents of any dynaset that was created using the parameter. The parameter's value is used only at the time that the SQL statement is executed, which is either at the **ExecuteSQL** call, the ODynaset **Open**, or the **ODynaset** Refresh. Finally, when we are finished with the parameter we can remove it, as in the example, with the **OParameterCollection::Remove** method.

Some details of this example are noteworthy. The **ODatabase** is passed into the ProcessCompany routine by address. It is legal to dereference an **ODatabase** address, and that is what we do here. It is also possible (and inexpensive) to pass the **ODatabase** by value. An **OField** object is used to get the department number out of the deptnumbers dynaset. An **OField** always gives you, from the current record, the value of the field to which it is attached. Therefore, as the dynaset is navigated, the **OField** objects returns different values. Because various cast operators are overloaded for the **OField** object, getting the value of the column is just a matter of casting the **OField** object.

Here is a more realistic version of the application from Example 7:

```
// transfer money from one account to another

// SQL statement to change a balance in the accounts table
static const char *setbalance =
   "update accounts set balance = balance + :amount where accno
= :anum";

void Transfer(ODatabase *bankdb,  // the bank database
             int debitaccount,   // account to debit
             int creditaccount,  // account to credit
             double amount)      // amount to transfer
{
   // we assume the existence of the parameters "amount" and "anum"

   // begin the transaction
   if (bankdb->GetSession().BeginTransaction() != OSUCCESS)
       return;  // couldn't start transaction

   // get the parameters
   OParameter amount = bankdb->GetParameters().GetParameter("amount");
   OParameter anum = bankdb->GetParameters().GetParameter("anum");

   // credit the first account
   anum.SetValue(creditaccount);
   amount.SetValue(amount);
   if (bankdb->ExecuteSQL(setbalance) != OSUCCESS)
   {
       bankdb->GetSession().Rollback();
       return;
   }

   // debit the second account
   anum.SetValue(debitaccount);
   amount.SetValue(-amount);
   if (bankdb->ExecuteSQL(setbalance) != OSUCCESS)
   {
       bankdb->GetSession().Rollback();
```

```
        return;
    }

    // it all worked
    bankdb->GetSession().Commit();

    return;
}
```

To use this routine, you open the bank database and set up the two parameters "amount" and "anum". Then for each transfer you call the Transfer routine. Notice that the actual SQL statement that is used is external to the routine. The SQL statement could actually be moved into some other code, making this Transfer routine even more general and maintainable.

Example 9: An advisory

Occasionally you need some piece of your code to be notified when something happens to a dynaset—either navigation or some editing. Typically the dynaset is controlled by one portion of your code and you monitor it with some other portion of your code (for example, some kind of general access management package).

You accomplish this monitoring with an advisory. Specifically, you attach an **OAdvise** object to an **ODynaset**. From then on, whenever something happens to the dynaset, messages are passed to the **OAdvise** object. The advisory can exert some control over the dynaset actions (for example, it can cancel them), and it can monitor what is happening.

The **OAdvise** class that is part of the Class Library does nothing. To get a useful advisory object, you must create a subclass of **OAdvise,** and then create an instance of that subclass. By overriding a few virtual functions, you can obtain the functionality you desire.

The *samples* directory contains a subdirectory called *posadv*, which contains the files POSADV.CPP and POSADV.H. These files implement a subclass of **OAdvise** called **PosAdvise**. The **PosAdvise** class keeps track of which record the dynaset is currently on— the first record being 0, the next 1, and so forth. It cancels the actions that it sees that would disturb its bookkeeping, such as a **DeleteRecord**.

Here's how you use **PosAdvise**:

```
// use of PosAdvise class

// open a dynaset
ODynaset dyn1;
dyn1.Open(db, "select * from emp");

// attach a PosAdvise advisory to the dynaset
PosAdvise dynposition;
dynposition.Open(dyn1);

// move to beginning of dynaset to get dynposition started
dyn1.MoveFirst();

// position the dynaset before we do some processing
PositionDynaset(dyn1);

// get the position
```

```
long startpos = dynposition.GetPosition();

// now do some processing
ProcessDynaset(dyn1);

// how many records did we process?
long nrecords = dynposition.GetPosition() - startpos;
```

The advisory is attached to the dynaset by using the **OAdvise Open** method. It is detached automatically when it is destroyed, or when **Close** is called.

Actions taken with the dynaset cause several things to happen:

- Before the action takes place, the **ActionRequest** methods of all attached advisories are called. The advisories can cancel the action at that point. (The **PosAdvise** class cancels all actions except **MoveFirst**, **MoveLast**, **MoveNext**, and **MovePrev**.)
- Next, if the action was allowed by all the advisories, it takes place.
- After the action, the **ActionNotify** methods of all the attached advisories are called.
- The **PosAdvise** class uses this notification to recalculate the position within the dynaset.
- Finally, the **PosAdvise** class can implement additional methods beyond the bounds of **OAdvise**. It implements the **GetPosition** method.

Example 10: OBound of a variable

All of our examples thus far have accessed the data in a dynaset more or less directly, either through **ODynaset** methods or using **OFields**. These are not always the most convenient access methods. Whenever we navigate to a new record and are interested in the values of various fields, we need to fetch the values explicitly. Further, when we want to edit a record or add records, there is a moderate amount of bookkeeping to do. It would be convenient to have a "managed dynaset"—an object that does the more routine work for us. This is the job of the **OBound** and **OBinder** classes.

Users of Visual Basic will note that the **OBinder** class plays the role of the data control (though without any user interface), and subclasses of **OBound** play the same role as bound controls.

You use an **OBinder** object much as you would use an **ODynaset** object. It is opened with database and SQL information, for example. It has methods that enable callers to add and delete records. In addition to the work that a dynaset can do, an **OBinder** object can keep track of changes that have been made to fields and can update the database automatically.

OBound objects hold the value of a single field of a record. When the dynaset changes, the value of the **OBound** instance is changed. And when the value of the **OBound** instance changes, the value of the corresponding field in the dynaset changes—with all the appropriate bookkeeping handled.

In the OMFC and OOWL libraries that are provided with the Class Library, you can see the use of **OBinder**s and **OBound**s. Those libraries contain machinery that creates normal widgets that are also subclasses of **OBound**. As a result, in a GUI program that uses these specially subclassed widgets, when the user edits the text of a widget and then navigates to another record, the database editing is done automatically (see Example 11).

In this example we subclass **OBound** so that we have an **OValue** variable that works with the **OBinder**-managed dynaset. The value of the bound **OValue** object is automatically set to the current value of the field with which it is associated. When the value of the **OValue** object is changed, the database is updated to reflect the change. We call this class **OBoundVal**. The implementation of this class is stored in the *samples* directory, in the subdirectory *boundval*. Our goal is to be able to write a program like this:

```
// silly example of use of OBoundVal
void GiveRaises(int minsalary, int saladd)
{  // give everybody with salary below minsalary a raise of saladd

   // open connection to database
   ODatabase odb("ExampleDB", "scott", "tiger");

   // construct the OBinder (the managed dynaset)
   OBinder block;

   // set up an OValue bound to the salary field
   OBoundVal salary;
   salary.BindToBinder(&block, "sal");
   // Note that we are binding the "sal" column before opening the
   // query. If the select list for the query does not contain the
   // "sal" column, the OBinder.Open() call will fail.

   // get the database data for the managed dynaset
   block.Open(odb,"select ename, sal, empno from emp order by empno");

   // If we do the binding here and the column we are attempting to
   // bind does not exist, the following call will fail:
   // salary.BindToBinder(&block, "sal");

   // Note that IsLast() will return TRUE after MoveNext() attempts to
   // move past the last record. Therefore the last record does get
   // processed in the loop
   while (!block.IsLast())
   {
       // check the salary of the current employee
       if ((int) salary < minsalary)
       {
           // change the salary
           salary = (int) salary + saladd;
       }
       block.MoveNext();
   }

   return;
}
```

The most interesting part of this sample is the line where the salary is changed. It appears that we're setting the value of a variable. However, we're actually changing the value of a field in the database. The database gets updated when we navigate to another record.

The **OBound** class has three very important methods:

1. **Refresh** is called (by **OBinder**) to give the **OBound** a new value whenever the field's value changes.
2. **SaveChange** is called (by **OBinder**) whenever the **OBound** should save its value to the database.
3. **Changed** is called by the **OBound** subclass implementation to notify **OBound** and **OBinder** that the value has changed.

Refresh and **SaveChange** must be implemented by any subclass of **OBound**. If the **OBound** subclass overrides any of the base class triggers steps, the overloaded triggers should call the default triggers in the base class to work correctly. Please see the online documentation for the **OBound** class for more details. The implementation for **OBoundVal** is shown below.

Refresh transfers the value from the database to the **OBound** object. **SaveChange** transfers the value from the **OBound** object to the database. **Changed** notifies the **OBinder** bookkeeping that a change needs to be saved to the database.

OBound is generally used as a "mix-in" class, adding functionality to some other class hierarchy. In this example we add functionality to the **OValue** class. **OBoundVal** multiply inherits from **OBound** and **OValue**.

Here is the **Refresh** method for **OBoundVal**:

```
oresult OBoundVal::Refresh(const OValue &val)
{
   OValue::operator=(val);  // use OValue assignment to get new value

   return(OSUCCESS);
}
```

When we are handed a new value, we use **OValue**'s assignment operator to save that value. This routine is called by the **OBinder** class. Here is the **SaveChange** method:

```
oresult OBoundVal::SaveChange(void)
{
   return(OBound::SetValue(*this));
}
```

We use the **OBound** helper routine **OBound::SetValue** to set the value. **SaveChange** is called by the **OBinder** class.

But how do we set the value of an **OBoundVal**? If we just allow the use of **OValue::SetValue**, our value will get changed but the **OBoundVal** class won't know about it. Further, we won't tell the machinery that there has been a change, so no change will get saved to the database. Here is the proper implementation for setting the value with an integer argument:

```
oresult OBoundVal::SetValue(int val)
{
   if (!Changed())
       return(OFAILURE); // couldn't start change
   OValue::SetValue(val);
   return(OSUCCESS);
}
```

The first thing we do is to call the **Changed**() method. (The default argument is TRUE, which indicates that we are making a change.) This sets a flag indicating that this **OBound** needs to have **SaveChange** called on it later, and telling the dynaset to attempt a **StartEdit**. This may fail, for a variety of reasons: the user may not have permission to edit this database, the record may be locked, and so forth. Only if the **StartEdit** is successful do we set the value of the **OBoundVal**, using the parent method **OValue::SetValue**.

We need to ensure that we override every **OValue** method that changes the object's value, because we need to call **Changed**() to make the **OBinder** bookkeeping aware of any change in the object's value. So we must also override the **Clear** and **operator=** methods. And, for extra utility, we also provide a number of extra **operator=** methods so that the **OBoundVal** can act like a regular variable as much as possible.

Note that when we use the **operator=**, we are copying the **OBoundVal**'s value, not the object itself. There are subtle differences. In general, subclasses of **OBound** want to copy values of objects rather than the objects themselves. For this reason the copy constructor and assignment operator are not implemented for **OBound**.

There isn't much difference in the way an **OBoundVal** is used and the way an **OField** object is used. Both return the field's value for the current record. Both enable you to set the field's value. However, the **OBoundVal** copies a value every time the dynaset moves to another record, but the **OField** does not. **OField** gets the value out of the dynaset only when it is requested. At the same time, If you use **OField** for editing, you must manage the **StartEdit**s and **Update**s of the dynaset yourself.

OBound becomes much more useful when the database fields are represented in a user interface. Then, the code for handling changes can be quite distributed, so keeping track of whether **StartEdit** has been called or not (for example) would be troublesome.

<u>**Example 11: A full emp table editor**</u>

This example demonstrates the construction of a complete application. We built the application using Microsoft's Visual C++ development environment, including the MFC framework classes. (Users of other development environments will be able to understand what was built from this example.) The sample source is in the EMPEDT subdirectory (emp table editor).

The example uses the connection dialog from Example 4, which provides a convenient interface for the user to log into the Oracle database. The main screen is a single window containing a form that displays and allows editing of all the fields of the employee table. We built it as an MFC formview. For editing the records of the table, we use the OMFC bound control classes (built using the **OBinder** and **OBound** machinery) provided with the Class Library. The example also includes a feature that generates a unique employee number automatically when new employee records are added. The amount of code needed to write this application is surprisingly small.

This example is broken up into several sections:

Application Layout	discusses the basic creation of the application
Form Creation	discusses the building of the main screen
Binding	discusses the **OBinder**-**OBound** hooks to the form
Buttons	discusses the implementation of navigation buttons
Using a Trigger	discusses the implementation of the new employee number feature
Error Handling	discusses the handling of errors

<u>Application Layout</u>

We generated the original application with AppWizard. We changed the view class to a form view so that a user interface could be constructed quickly with the App Studio resource editor. The application's memory model needs to be set to large.

We set the project to link against the oraclm, omfc, ole2, and ole2disp libraries. Although you don't need to interact with OLE to use the class library, the class library uses OLE internally, so it needs to be linked.

We use the connection dialog from Example 4. We added the files LOGDLG.CPP and LOGDLG.H to the project. Then we copied the dialog resource IDD_LOGIND from the LOGDLG.RC resource file to the EMPEDT.RC resource file.

We made one simple change to the application class: we placed the class library initialization call (**OStartup**) in CEmpedtApp::InitInstance. We added a destructor method for the application class (~CEmpedt) and placed in it a call to **OShutdown**.

Form Creation

The purpose of this application is to edit the *emp* table—one of the standard Oracle demonstration tables. The emp table has eight fields: ename (employee name), empno (employee number), job (a keyword describing the employee's job), mgr (the employee ID of the employee's manager), sal (the employee's salary), comm (a commission to be paid to the employee, which is often NULL), hiredate (date that the employee was hired) and deptno (identification number of the employee's department—used to reference the *dept* table).

We built the most straightforward kind of form to edit this table: one text edit per database field. We edited the form in App Studio. We gave each edit control a unique ID. We also placed a static text item next to each edit control as a label. The IDs of the edit controls, their order, and what the label says or what relation it has to the control is completely up to you. Each of the edit controls will be attached to a database field by code that you write, instead of based on assumptions that the Class Wizard might make.

Neither variables nor message maps were created for the edit controls. The edit controls will be completely managed by the **OBoundEdit** class.

We added a button called Connect to the form. The user presses the Connect button to connect to the Oracle database. We used Class Wizard to create a button-clicked method for the connect button. That method is named OnConnect.

Binding

The form view class CEmpedtView controls the operation of the form view. We added a single **OBinder** instance as a member variable to the view class. Its name is m_empblock. (It is actually declared as **OBinderEmp**. **OBinderEmp** is a subclass of **OBinder**. See *Error Handling* below.) That **OBinder** instance manages the database connection, the dynaset, and all the bookkeeping needed for editing the database data. Next we need an **OBound** subclass instance for each of the user-interface widgets that we want bound to a database field. Since edit controls were used for the user interface, we used **OBoundEdit** instances. We added one member variable of type **OBoundEdit** to CEmpedtView for each edit control. (The member variable for empno is an instance of a subclass of **OBoundEdit**; see *Using a Trigger* below).

The edit controls are bound to database fields by calls to methods in the **OBound** class and subclasses. This binding is done after a database connection is established. For simplicity, we placed all the binding code in the **OnConnect** method.

The first thing that the **OnConnect** method does is to connect to the database. It does this by calling the login dialog class:

```
    // get an ODatabase object via the connection dialog
    logdlg connd;

    // get a database object
```

```
      ODatabase odb = connd.GetLogin(ODATABASE_PARTIAL_INSERT |
ODATABASE_EDIT_NOWAIT);
     if (!odb.IsOpen())
      { // didn't get a connection - user must have canceled
          return;
      }
```

The login dialog prompts the user (via a dialog) for database name, user name, and password and attempts to connect to the Oracle database. When it is successful it returns an open **ODatabase** object that refers to the Oracle database. Unsuccessful connection attempts put up an error dialog. If the user cancels the login dialog, an unopened **ODatabase** object is returned. So **OnConnect** checks the returned **ODatabase** to see whether it is opened. This amounts to a check whether the user canceled from the login dialog or connected to the Oracle database successfully.

The database is opened with partial insert and nowait options. The partial insert option ensures that any default field values set by the database are properly reflected in the dynaset when a record is added or edited. The nowait option ensures that if another user has a lock on a record that we wish to edit, we don't wait for that lock to be freed. Instead we get an error.

The **OBoundEdit** controls need to be attached two ways. First, they need to be attached to particular fields that will be available in a particular **OBinder**. Second, they need to be attached to a particular user interface widget. We accomplish the first attachment with a call to **OBound::BindToBinder**:

```
      m_ename.BindToBinder(&m_empblock, "ename");
```

This call binds the m_ename member variable to the "ename" field available in **OBinder**. The ename field becomes available once the binder object (m_empblock) is opened. It is legal to bind a control to a binder before the binder's SQL statement has been executed.

We accomplish the second attachment with a call to **BindToControl**, which is a method available for the classes in the OMFC and OOWL libraries:

```
      m_ename.BindToControl(this, IDC_ENAME);
```

This call binds the m_ename member variable to the user-interface widget identified by IDC_ENAME in the window identified by "this". Since **OConnect** is a method of the formview class, "this" points to the form window.

Once all the **OBoundEdit** controls are hooked up the **OnConnect** method, we can open the **OBinder** by executing an SQL statement:

```
      m_empblock.Open(odb, "select * from emp order by empno");
```

This statement creates a dynaset, fetches records from the database, and puts the correct database field values in all of the edit controls.

<u>Buttons</u>

With the login dialog and the edit controls implemented, we can build and run the application. It works; it shows the values of the first record. We added more buttons to the form to implement additional functionality.

We implemented navigation by adding buttons named First, Prev, Next, and Last. We used Class

Wizard to create a button-clicked method for each. The implementation for each is merely one line of code. For instance, to do Next:

```
        m_empblock.MoveNext();
```

Calling **OBinder::MoveNext** moves the dynaset to the next record, fetches it from the database (if necessary), and updates the values of all the edit controls to display the new record. It actually does more than that. If any of the edit controls had been used to change the data in a record, **OBinder::MoveNext** saves that change to the database before moving to the next record. All this is accomplished without writing anything more than "MoveNext".

We added a few more buttons for record-level edits: Add New, Duplicate, and Delete. These were also very simple to implement, a single line of code for each. Again, the **OBinder** class takes care of necessary bookkeeping.

Using a Trigger

The empno field of the emp table is required to contain a unique number in each record. Rather than forcing the user to create the unique number manually every time a new record is added (with either Add New or Duplicate), we added code to set the unique number for the user.

Calculating a unique number is not difficult. One way is to ask the Oracle database the current maximum value of empno across all records. Adding one to that result guarantees a unique value, relative to the records currently in the database. (A better way is to use a sequence. Consult your Oracle documentation for more information about sequences.).

When should this calculation be made and what should you do with the result? The calculation needs to be made every time a new record is added. We could do the calculation every time the Add New or Duplicate button is pressed, and then change the value of the empno edit control manually. Then we would have to call **OBound::Changed** to make sure that the **OBoundEdit** instance knew that the value had changed, so that the **OBinder** would update the database properly. This approach would work, but it is prone to error. For example, what happens if we add some other method that also adds a record?

Instead, we changed the behavior of the empno edit control. Whenever a new record is added to an **OBinder** dynaset, every **OBound** instance bound to that **OBinder** calls its **PreAdd** method before the addition and the **PostAdd** method after the addition. The **PostAdd** method of **OBoundEdit** does nothing. We subclassed **OBoundEdit,** , creating a new class called **OBoundEmpnoEdit**, and defined a **PostAdd** method for it that overrides the default **PostAdd**. We declared the empno member variable as an instance of **OBoundEmpnoEdit**. **OBoundEmpnoEdit** is a subclass of **OBoundEdit** and is implemented in EMPEDVW.H and EMPEDVW.CPP.

At runtime, whenever a new record is added to the dynaset, the **OBoundEmpnoEdit::PostAdd** method is called after the record is added. That routine sets the value of itself, which sets the value of the edit control to the unique number that we have calculated.

Because the trigger methods are all virtual, the **OBinder** manages the **OBoundEmpnoEdit** as if it were a normal **OBound**. But we have overridden the default **OBound** behavior to suit the needs of the application.

Although in this particular sample **OBoundEmpnoEdit** doesn't need any information from the rest of the program, in a real working program the operation of the bound control would depend on whatever else is going on. The bound object needs a context. Just to show one way you can accomplish that, the sample code also implements a **SetContext** method that hands some useful context to the bound object. That context would then be available in its methods for use.

Error Handling

The sample application performs some minimum error handling. When an error occurs, it gives the user an error message.

The first kind of error that can occur is that the class library or some component upon which it depends cannot be loaded or initialized. This error is caught just before we try to log onto the database. We use these lines of code:

```
// get the default session
   OSession defsess(0);

   if (!defsess.IsOpen())
   { // couldn't get default session?  Class library isn't working
       AfxMessageBox("Can't start Oracle class library");
       return;
   }
```

If the class library can't be initialized, then the default **OSession** object won't be able to Open.

The next kind of error that can occur is that the **OBinder** object can't be opened. We use this code:

```
      // check for error
      if (ores != OSUCCESS)
      { // we couldn't open the dynaset
          const char *msg;
          if (defsess.ServerErrorNumber() != 0)
          { // we have a server error - tell the user that
              msg = defsess.GetServerErrorText();
          }
          else
          { // no server error - class library isn't working correctly
              msg = "Class library error when opening dynaset.";
          }

          // give the user a message
          AfxMessageBox(msg);
      }
```

where ores is the result of the **OBinder::Open** call. The most likely errors here are that the SQL statement has an error (the table doesn't exist or one of the field names is bad) or that the user doesn't have permission to read the table. These are both Oracle errors.

Either of these two cases would prevent the application from working at all. These errors are generally bugs in the client program that can be avoided with correct coding (getting the SQL statement right, for instance). Once we've opened the **OBinder** , we will look at records in the table. Then we have to start considering other types of errors—interactions with other users or lack of system resources.

The most common error occurs when we attempt to edit the value of a field. When we try to change the value of the field, the **OBinder** instance attempts a **StartEdit** on the dynaset. This can fail for a variety of reasons. Most common are: connection to the server is lost, another user has a lock on the row, or the data in the row has been changed by another user. The first two are reported as Oracle errors. The last is an error specifically generated by the class library to ensure that database data is not accidentally overwritten.

The difficulty here is that, with **OBoundEdit** controls, the data is changed by typing a key into the edit control. Our program isn't making any method calls whatsoever. How do we get control to report the error?

The **OBinder** class has a method called **OnChangedError** that can be overridden. It is called when an error occurs while processing the **StartEdit** on the dynaset. The example uses this to inform the user of an error when a record is starting to be edited. The **OBinder** class is subclassed to **OBinderEmp,** and the member variable is declared to be of that type. Then when an error occurs as a user types a value into a record, the **OBinderEmp::OnChangedError** routine is called.

Finally, errors can occur during the various operations represented by the buttons on the form: navigating to various records or adding and deleting records. Errors during these operations are all sent to the view **HandleError** method. **HandleError** takes care of errors that occur when navigating, when adding or deleting records, or when saving a changed record to the database. Any of the navigation methods, or any method that adds records, saves any changes to the current record before doing its work (this is actually done through PreOperation trigger functions). **HandleError** reports the error to the user by fetching (or calculating) the correct error message.