OOWL BOUND WIDGET LIBRARY

This file documents the OOWL library provided with the Oracle Objects for OLE C++ Class Library. The OOWL library of classes are used to build GUI programs using Borland's OWL framework. These classes have been built using Borland C++ 4.0 and 4.5.

Differences between Borland C++ 4.0 and 4.5

With the release of version 4.5, Borland made some minor changes to the implementation of OWL. This problem manifests itself with an undefined symbol linker error when attempting to link an application using Borland C++ 4.5 with the bound widget library (***oowl.lib***) created using Borland C++ 4.0 that was part of the version 1.0 of Oracle Objects for OLE. This release contains a second library called ***oowl45.lib*** that was created using Borland C++ 4.5. You will find it in the same directory as the original ***oowl.lib*** (**[ORACLE_HOME]**\oo4o\cpp\owl\lib - where **[ORACLE_HOME]** is the directory you have installed your Oracle products for Windows).

If you are using Borland C++ 4.5 and get the following Linker Error:
Undefined symbol v_U_U_W_Dispatch(GENERIC far&,void(GENERIC::*) (unsigned int,unsigned int,const HWND__ near*),unsigned int,long);
You should edit the project (IDE) file and change the entry for ***oowl.lib*** to ***oowl45.lib***. You may also need to edit the project path information for the sample application (see below) in order to make it compile and link correctly.

The Borland C++ 4.5 compiler also puts references into the Class Library DLL to redistributable Borland DLL's that changed from version 4.0. For example, OWL200.DLL changed to OWL250.DLL in version 4.5. If you have Borland C++ 4.5, you should use oraclb45.dll and the import library oraclb45.lib. They are installed into the same directories as the original 4.0 versions of these files.

How to build the sample program.

The sample program can be found in the **[ORACLE_HOME]**\oo4o\cpp\owl\samples\empedit directory (where **[ORACLE_HOME]** is the directory your Oracle products are installed - usually c:\orawin). The project file is bltest.ide and can be loaded by Borland C++ version 4.0 and 4.5. This project file contains paths to the various header (*.h) and library (*.lib) files needed to compile and link the sample program. Unfortunatly, you may have installed Borland C++ in a different directory (or even a different drive) and the project path information will not be correct.
To change the path information, select Options from the IDE main menu. Select Project and then select Directories from the *topics* list. The dialog will now show some edit controls containing the include and library search paths. There are four components to the include path, separated by semicolons. The first is for the current directory and can be ignored. The second is the Borland C++ include files which is set to \bc4\include. You may need to change this to the directory in which Borland C++ is actually installed - if you are using version 4.5, this will probably be \bc45\ include. The third and fourth components are relative paths to the Oracle Objects include files. You should not need to change these unless you have moved any files from the original directory tree. If you have moved the project or any header or library files, you will need to alter these paths. Specifically, you will need to add references for the class library include files and the oowl include files. These are installed into **[ORACLE_HOME]**\oo4o\cpp\include and **[ORACLE_HOME]**\oo4o\ cpp\owl\include respectively. The second edit field contains the library path and should be modified to contain the paths to ***oraclb.lib*** and ***oowl.lib*** (or ***oraclb45.lib*** and ***oowl45.lib***). These are installed into **[ORACLE_HOME]**\oo4o\cpp\lib and **[ORACLE_HOME]**\oo4o\cpp\owl\lib respectively.

The library search path may also need to be changed to include the lib subdirectory of the Borland C++ installation (e.g. d:\bc45\lib). Please remember you will need to specify the full path (including the drive letter) for any component that is on another disk drive from the one you are

running the project from.

In order to run the sample (or any application you build), it will need to access the Class Library runtime DLL - ***oraclb.dll*** (***oraclb45.dll*** if you are using Borland C++ 4.5). The easiest way to do this is to copy the DLL to a directory on your path (such as \orawin\bin) or your windows system directory. The latter is recommended. You will also need to include the correct import library for the Class Library DLL in your project. They are in **[ORACLE_HOME]**\oo4o\cpp\lib and are named ***oraclb.lib*** and ***oraclb45.lib*** respectively. You can find the DLLs in **[ORACLE_HOME]**\oo4o\cpp\bin, or if you want to debug in the class library, you should use the versions in **[ORACLE_HOME]**\oo4o\cpp\bin\dbg. Debuggable versions of ***oowl.lib*** and ***oowl45.lib*** are also supplied in **[ORACLE_HOME]**\oo4o\cpp\owl\lib\dbg. We have noticed that the integrated debugger cannot always read the DLL's symbol table correctly unless the DLL you are debugging is in the *current directory*. This is usually the same directory as the executable that calls the DLL.

The sample application uses an enhanced version of the emp table that is part of the demonstration database. The enhanced table is called emp2 and can be created by running demoemp.sql which is installed into **[ORACLE_HOME]**\oo4o\cpp.

What problem do these classes solve?

The basic classes of the Oracle Objects for OLE C++ Class Library enable you to access the data in an Oracle database. You can fetch records, add records, edit records, execute arbitrary SQL statements, and so forth. However, if you want to write a GUI program that displays the database data, you are on your own. You must fetch the data, push the data into your GUI widgets, and repeat whenever the dynaset moves to another record. If the widget is used to edit the data you must execute a StartEdit, SetValue, Update cycle.

The **OBinder** and **OBound** classes make this work much easier. An **OBinder** instance manages a dynaset. **OBound** instances are attached to fields in the **OBinder**'s dynaset and "bound" to the **OBinder** instance. From then on the **OBinder** and **OBound** code do most of the tedious bookkeeping for you: the **OBound** instance values are changed when needed, and when edits are made through the **OBound** instances they are saved to the Oracle database.

The C++ Class Library provides an implementation of **OBinder**. However, it provides only a pure virtual **OBound** class. To make use of the convenience of **OBinder**, you need subclasses of **OBound** that implement the **OBound** functionality.

The classes in this OOWL library are subclasses of **OBound**. They provide GUI widget implementations of the **OBound** functionality. As a result you can create a form using Borland C++'s Resource Workshop resource editor and, with very few lines of code, you can hook those widgets to database fields. You then have a working application.

Please see the *Oracle Objects for OLE C++ online help system* for more discussion of **OBinder** and OBound.

Note to Visual Basic users: An **OBinder** object works like a data control (it has no user interface but it performs all the bookkeeping). **OBound** objects work like bound controls.

What kind of objects are available?

There are classes for the following kinds of user interface widgets:

edit controls:	display (and edit) values as strings
static text controls:	display values as strings

check box:	display (and edit) values as "on" or "off"
radio button:	display (and edit) values as a radio button
slider:	a combination slider and text display that graphically displays and edits a numeric value, either horizontal or vertical
gauge:	graphic display of a numeric value (read-only)

How are these objects used?

This library (and in fact the entire C++ Class Library) is used to build large model programs.

To make and set up instances of any of these classes, you must go through several steps:

1. First, you must create the user interface widget. In Borland C++ this is easiest to do using Borland's Resource Workshop resource editor. You can also create the widget programmatically (constructors are provided for this purpose).

2. Next, you must declare an **OBinder** instance in your application, typically in the view class for the window where the database form resides.

3. You must declare an **OBound** subclass instance for each widget. These instances are usually members of the view class for the window where the database form resides. Usually your instance variables are pointers to widget objects. You call new and construct the instances in the view's constructor.

4. You must call the "**BindToBinder**" method on each of the **OBound** subclass instances. You can do this multiple times, but normally you do it just once (for example, in the "**SetupWindow**" method of your view). The **BindToBinder** method tells the instance variable what **OBinder** to get its data from and what field it should access in the **OBinder**'s dynaset.

5. The final step in setting up the instance is to open the **OBinder**. This creates a dynaset and fetches records from the database.

   Your application can now run. You need to implement some way for the user to navigate through the records of the dynaset. For example, you can create a button with the label "Next" that calls **OBinder::MoveNext**. The user can make changes in the widgets. Just before the dynaset is navigated to another record (for example, in response to a **MoveNext** call), all changes on the current record are saved.

7. Finally, when the program is exiting, it is a good idea to call **OBinder::Close** explicitly This is not strictly necessary, but is good form. See **OBinder::Close** in the *Oracle Objects for OLE C++ online help system*.

An example

An example that uses the OOWL classes is provided. When you installed Oracle Objects for OLE, the sample was placed in the *Samples* subdirectory of the *OOWL* directory (if you asked to install sample code). The example allows the user to edit the emp2 table (which can be created with the DEMOEMP.SQL script), which is an extended version of the sample emp table provided with Oracle databases. The interesting files are BLTSTDLG.H (which declares the bound control variables for the dialog) and BLTSTDLG.CPP (which uses the bound controls).

Methods of all the classes

The methods described below are available in all the classes. The methods that are inherited from **OBound** are not documented here (**BindToBinder** in particular). Please see the *Oracle Objects for OLE C++ online help system* for more information on **OBound** and its methods.

operator= and copy constructor

All of these classes define the assignment operator and copy constructor in the class definition but do not implement them. This prevents the use of the compiler's default assignment operator (or copy constructor), which would be wrong. Neither the assignment operation nor construction by copy is defined for any of these classes. If you inadvertently use assignment or copy construction on one of these objects, you get a link error.

The individual classes

OBoundEdit

This is the class you will most often use. It displays the database value in a text Edit control. It has the additional method:

OBoundEdit

Usage: OBoundEdit(TWindow* parent, int Id, const char far* text, int x, int y, int w, int h, UINT textLen = 0, BOOL multiline = TRUE, TModule* module = 0)
OBoundEdit(TWindow* parent, int resourceID, UINT textLen = 0, TModule* module = 0)

The arguments to the constructors are the same as for the **TEdit** constructor.

SetProperty

This method specifies whether the control is read-only or read-write.

Usage: oresult SetProperty(bool *mode*=OBOUND_READWRITE);

mode a flag indicating whether the control is read-only or read-write. It can be either:
OBOUND_READWRITE
OBOUND_READONLY

OBoundStatic

OBoundStatic

usage: OBoundStatic(TWindow* parent, int Id, const char far* title, int x, int y, int w, int h, UINT textLen = 0, TModule* module = 0);
OBoundStatic(TWindow* parent, int resourceId, UINT textLen = 0, TModule* module = 0);

The arguments to the constructor are the same as for the **TStatic** class. Widgets of this class are always read-only.

OBoundCheckBox

This class allows you to display and edit database values as a checkbox. It is of greatest value for a database field that only has two possible values, such as TRUE and FALSE. The **SetProperty** method enables you to specify what value will be considered "on" and what value will be considered "off". When the user checks the checkbox, the field data is set to the "on" value. When the user unchecks the checkbox, the field data is set to the "off" value.
If the field value is neither "off" nor "on", the checkbox behaves as follows:

If the checkbox is tristate, it is placed into the "grayed-out" state.
If the checkbox is not tristate, it is off.

OBoundCheckBox

Usage: OBoundCheckBox(TWindow* parent, int Id, const char far* title, int x, int y, int w, int h, TGroupBox *group = 0, TModule* module = 0)
OBoundCheckBox(TWindow* parent, int resourceID, TGroupBox *group = 0, TModule* module = 0)

SetProperty

This method specifies whether the control is read-only or read-write, and what the "on" and "off" values are for the checkbox.

Usage: oresult SetProperty(const OValue *onvalue*, const OValue *offvalue*, bool *mode*=OBOUND_READWRITE)

onvalue the value corresponding to the checkbox being checked
offvalue the value corresponding to the checkbox being unchecked
mode a flag indicating whether the control is read-only or read-write. It can be either:
OBOUND_READWRITE
OBOUND_READONLY

OBoundRadioButton

A single radio button cannot represent a database field value. A group of radio buttons can represent a value. Each radio button corresponds to a single possible value, and the single radio button that is on indicates the actual database field value. Choosing a different radio button changes the field value.

Before constructing the **OBoundRadioButton**, you may need to construct a **TGroupBox** object to group the radio buttons. This works the way the **TRadioButton** class works.

If the current field value does not correspond to the value of any of the radio buttons in the group, none of the radio buttons are selected.

OBoundRadioButton

Usage: OBoundRadioButton(TWindow* parent, int Id, const char far* title, int x, int y, int w, int h, TGroupBox *group = 0, TModule* module = 0)
OBoundRadioButton(TWindow* parent, int resourceID, TGroupBox *group = 0, TModule* module = 0)

The arguments to these constructors are the same as for **TRadioButton**.

SetProperty

This method specifies whether the control is read-only or read-write, and what value this radio button represents.

Usage: oresult SetProperty(const OValue *value*, bool *mode*=OBOUND_READWRITE)

value the value that this radio button represents
mode a flag indicating whether the control is read-only or read-write. It can be either:
OBOUND_READWRITE
OBOUND_READONLY

OBoundHSlider

A slider is a combination of a scroll bar. It is suitable for displaying and editing numeric data that has a known range. The data is displayed and edited using the scroll bar.

You must create **OBoundHSlider**s programmatically.

OBoundHSlider

Usage: OBoundHSlider(TWindow* parent, int Id, int x, int y, int w, int h, TResId thumbResId = IDB_HSLIDERTHUMB, TModule* module = 0);

The arguments to this constructor are the same as for **THSlider**.

SetProperty

This method specifies whether the control is read-only or read-write, sets the range of the scroll bar.

Usage: oresult SetProperty(const OValue &*minvalue*, const OValue &*maxvalue*, bool *mode*=OBOUND_READWRITE)

minvalue the minimum value of the scroll bar
maxvalue the maximum value of the scroll bar
mode a flag indicating whether the control is read-only or read-write. It can be either:
OBOUND_READWRITE
OBOUND_READONLY

OBoundVSlider

This class is the same as **OBoundHSlider**, except that it is vertical instead of horizontal. See the **OBoundHSlider** documentation above.

OBoundGauge

This class provides you with a gauge display of a database field value. It is suitable for displaying numeric data with a known range. It is always read-only. You must create **OBoundGauge** programmatically.

OBoundGauge

Usage: OBoundGauge(TWindow* parent, const char far* title, int Id, int x, int y, int w, int h, BOOL isHorizontal = TRUE, int margin = 0, TModule*module = 0)

The arguments to this constructor are the same as for **TGauge**.

SetProperty

This method sets the range of the gauge.

Usage: oresult SetProperty(const OValue &*minvalue*, const OValue &*maxvalue*)

minvalue	the minimum value of the scroll bar
maxvalue	the maximum value of the scroll bar