Putting OWL 2.0 Through its Paces

Ted Faison

Ted is a writer and developer, specializing in Windows and C++. He has authored several books and articles on C++, and has been programming with the language since 1988. He is president of Faison Computing Inc, a firm which develops C++ class libraries for DOS and Windows. He can be reached at tedfaison@msn.com.

Introduction

When OWL 1.0 was released back in November 1991, it represented a quantum jump for C++ developers of Windows applications. The learning curve was very steep, but the benefits substantial. OWL 1.0 enabled project development times to be reduced from months to weeks, spawning a whole new generation of applications, well recognized by their use of the so-called *Borland style* controls, with the *chiseled-steel* dialog boxes. Version 2.0 of OWL, although not revolutionary, adds many features that further simplify application development. Support has been added for custom controls, dialog boxes as main windows, tool bars, full-fledged status lines a la Word for Windows, GDI support (with classes for pens, brushes, regions, fonts, palettes, cursors, bitmaps, icons and DIBs), printer support, and more. In this article, I'll explore the most interesting new features of OWL 2.0. Before diving into the new features of OWL 2.0, I'll spend a few words on OWL 1.0, to see why Borland chose to change course with 2.0, resulting in a product that is incompatible with OWL 1.0. For the record, this article is based on a prerelease (gamma) version of OWL 2.0, so the shipping version of OWL may be a little different from what I worked with.

The limitations of OWL 1.0

OWL 1.0 was developed in a time when C++ libraries were few and their development heavily influenced by the work of the Smalltalk community. Indeed, OWL 1.0 was built around a class hierarchy that was very Smalltalk in style: at the root of the hierarchy was a class named *Object*, and almost no use was made of C++ features like virtual base classes or multiple inheritance. What OWL 1.0 unfortunately didn't borrow from Smalltalk was the *Model View Controller* (MVC) paradigm, which defines a separation between the processes of managing a window's basic data, representing the data on the screen, and interacting with the end user of a program. OWL 1.0 tended to make all the code for handling and displaying a window wind up in a single class -- typically derived from a class called `TWindow` -- resulting in a very tight coupling between the model data and the way the data was displayed.

Persistence appeared to be added as an afterthought to OWL. In fact, persistence is really the only part of OWL 1.0 that seems to be implemented in C++ style. The only time multiple inheritance is used in OWL 1.0 is in support for persistence, with the multiply inherited base class `TStreamable`. The `pstream` hierarchy, to handle persistent streams also uses multiple inheritance, and has an `iostream` flavor to it. The `pstream` hierarchy has several classes that correspond directly to `iostream` classes. For example, the classes `fpstream`, `ifpstream` and `ofpstream` are equivalent to the `iostream` classes `fstream`, `ifstream` and `ofstream`. Manipulators are used in the `pstream` hierarchy to insert and extract objects from `pstreams`, using the same notation as with ordinary `iostreams`.

The main problem with OWL 1.0 was that it wasn't created by Borland but by another company, namely The WhiteWater Group, which had originally developed the Actor programming language for Windows [see MSJ Vol 4 No 2, March 1989]. Actor was something of a cross between Smalltalk and C++, developed specifically for Windows. WhiteWater accumulated substantial experience with Windows class hierarchies, and developed OWL in C++, which they then licensed Borland. OWL 1.0 did give Borland a

jump start into the C++ Windows application frameworks market, but it didn't give them everything they wanted. What they didn't get was a library that was really C++ in style and rich in functionality. Most complex C++ class hierarchies have more than one root class, whereas OWL 1.0 had the single root `Object`, taken directly from Smalltalk. No virtual base classes were used, so you couldn't easily create a class that was multiply derived from OWL 1.0 classes. Even if you succeeded, the resulting class could not be made persistent. Also, the OWL 1.0 containers (like `TNSCollection` and `TNSSortedCollection`) were not very powerful, not based on template classes, and incompatible with both the `Object`-based containers and the BIDS container classes. Here is a list of the major areas in which OWL 1.0 was deficient:

- GDI support
- Support for toolbars, tool palettes and status bars
- Use of Containers
- Model / View structure
- Printer support
- DDE/ OLE support
- Clipboard support
- Data Validation support

Even with its limitations, OWL 1.0 was still a remarkably useful product, because it was an open one. Almost all the member functions in the library were declared either `protected` or `public`, allowing derived classes to change even the most basic features. For example, using a dialog box inside MDI child windows hadn't been contemplated in the original design, but was relatively easy to implement by adding a little code in a derived class to change the way accelerator keys were handled. Anything not supported by OWL directly could easily be added by programmers, either through custom classes or direct Windows API calls.

From 1.0 to 2.0

One of the biggest problems Borland faced in the marketplace with OWL 1.0 was over the use of a C++ language extension made to support the binding of C++ member functions to Windows messages. The extension involved the declaration of functions known as *Dynamically Dispatched Virtual Table* (DDVT) functions. Although elegant, the extension was completely non-standard. In 1991 the non-standard feature didn't raise that many eyebrows, but in today's market with increasing competition between Microsoft, Borland and Symantec in the C++ arena, things are different. One of Borland's goals with OWL 2.0 was to create a standard product that can be used with any ANSI-compliant C++ compiler. Borland would love to have Visual C++, Watcom and Metaware C++ programmers all use OWL 2.0. To this end, the Borland developers made some rather dramatic changes to the product.

One of the first things Borland got rid of were the DDVT functions, which were replaced with MFC-style messages maps called response tables. This immediately caused any code written for OWL 1.0 to be incompatible with OWL 2.0. Borland also revamped OWL to take advantages of the new features that were incorporated into the ANSI draft of C++ since development of OWL 1.0 started. Among the new features, support of the new standard **string** class, the use of templates for containers, and exception handling. Borland decided that rather than perpetuating a design that had basic problems, in the name of compatibility - they would bite the bullet and make OWL as good as they possibly could, in terms of the structure of the class hierarchy. The designers probably could have gone a little further to support features like memory leak detection and support for OLE 2.0 and ODBC, which are supported in MFC 2.5. My guess is that Borland was forced to leave them out to get the product to market in a timely manner.
As a result of all Borland's changes and improvements, OWL 2.0 is compatible with Windows 3.1, Win32s and Win32. OWL is therefore a good tool for implementing 16 or 32 bit applications, and the new OWL is a much better product and much richer than OWL 1.0 -- with full compliance with the ANSI draft. OWL has dozens of new classes, and has a rather extensive class hierarchy, as shown in figure 1.

See the figure on Pages 1-2 of the Borland Beta Documentation *OWL 2.0 Reference Guide*.

Figure not available

Figure 1 - The complete OWL 2.0 class hierarchy.

As you can see in figure 1, OWL is actually made up by several smaller class hierarchies. There is the document/view hierarchy, the child control hierarchy, the data validator hierarchy, the device context hierarchy, the GDI class hierarchy, the gadget hierarchy and so on. Each of these is described in the following sections. OWL use multiple inheritance to create a number of classes, such as `TWindow`, `TDocManager` and `TView` (all derived from `TStreamable` and `TEventHandler`), `TWindowView` (derived from `TWindow` and `TView`), `TDecoratedFrame` (derived from `TFrameWindow` and `TLayoutWindow`), etc. OWL uses virtual base classes so allow you to create new classes that are multiply derived from OWL classes, without incurring problems with multiple copies of the same base class.

Basic OWL Features

Let's get to the details. For starters, OWL 2.0 - like MFC 2.5 - follows the MVC paradigm. Borland uses a refinement of the original MVC paradigm, and calls it the Doc/View Model. Borland loosely uses the term *document* (as does Microsoft in expressions like *Multiple Document Interface)* to refer to generic classes that handle data. The term document is used regardless of whether the data in question is a word-processing document, a multi-media object, a bitmap or other.

There are several advantages in separating classes that manage data from the classes that display it. The first benefit is greater simplicity. One class has access functions to read and write values, another class presents data on the screen, formatted appropriately. The screen can be designed independently of the way data is stored or managed by the underlying data model. Changes in the screen layout have no effect on the model, and vice versa.

Another advantage is that the same data can be displayed differently by different viewers, or the same viewer can be used in multiple windows, to show different parts of the data. For example, the Windows File Manager is an MVC application, in that it displays files using two windows: one shows only directories, the other only files. Multiple windows can be opened, each showing the files in a different directory, and the user of the program can modify the way the files are shown in the screen. There is only a single instance of the data (the files on a disk), but multiple, independent views.

Figure 2 shows a portion of the OWL 2.0 class hierarchy, related to viewer classes and document classes:

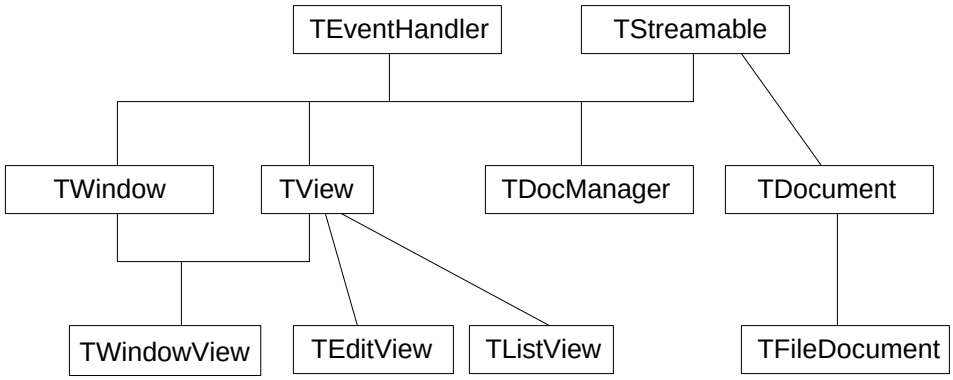

Figure 2 - The OWL 2.0 class hierarchy supporting the Doc/View model.

I'll describe the main parts of the Doc/View model shortly. Class `TWindow` is the base class for all other windows, including frame windows, dialog boxes and controls. OWL has a number of specialized classes to handle MDI and SDI applications, also allowing you to easily use a dialog box as a main window. Custom controls are supported through class `TControl`, derived from `TWindow`. OWL also supports common application features as status bars, message bars, tool bars and floating tool palettes. The Microsoft Common Dialog Boxes are also supported by specialized OWL classes, reducing the amount of code required by applications.

Documents

In OWL's Doc/View model, documents are classes that handle your application's underlying data. Documents handle all kinds of data -- not just character documents. Documents can be compound, by including other documents as sub-documents, allowing documents to be arbitrarily complex.

`TDocument` is the base class for all document objects. Documents are designed to be persistent, although the details of inserting and extracting a document's data to a stream are entirely application-dependent. Class `TDocument` has a number of virtual functions that support persistent stream operations. Derived document objects are required to override these functions to read and write whatever data they require.

When you develop an OWL application, your aren't required to use documents or views. Simple applications can be developed using the traditional approach: everything is a single class or class hierarchy. The Doc/View model is useful to handle more complex situations -- especially MDI applications. AppExpert (see the sidebar for an overview of AppExpert) allows you to generate applications using both a Doc/View approach, or the simpler SDI/MDI one.

Views

Documents have no way of interacting with the user of an OWL program. A document without a viewer is about as useful as a boat anchor in the Sahara desert. View objects (also called *viewers*) allow a document's data to be presented to the user. Viewers know how to take document data and format it for presentation. One viewer can be connected to only one document, but there can be unlimited combinations of documents and viewers. Viewers display data by connecting two other objects together: a document and a window. The document knows nothing about the window and vice versa. Only the viewer knows what's really going on.

OWL has three built-in viewers, to handle the most common situations. The viewers are implemented in the classes `TEditView`, `TListView` and `TWindowView`. `TEditView` presents a document's data in a simple editor window that uses a maximized Windows Edit control. TEditView only handles unformatted text files. `TListView` is like `TEditView`, except it uses a maximized Windows Listbox to display text information. `TWindowView` is the plain vanilla viewer, designed to be used as a base class for application-dependent viewers, such as TIFF file viewers, hex file viewers, etc.

Document Templates

Documents and viewers don't live in isolation. With OWL 2.0, documents are associated with one or more viewers. The association is made through a template class, using the macro DEFINE_DOC_TEMPLATE_CLASS. When I first saw this macro, my reaction was, "not another macro! Macros are for C programs...". Although not in my style, the macro does simplify the connection between

a document and a view. In fact the macro handles all the details. To setup a Doc/View template, you do something like this:

```
DEFINE_DOC_TEMPLATE_CLASS(TSomeDocument, TSomeView, TMyTemplate);
```

This creates a new type called `TMyTemplate`. The macro DEFINE_DOC_TEMPLATE_CLASS is rather elaborate, expanding into the following code:

```
#define DEFINE_DOC_TEMPLATE_CLASS(docClass, viewClass, tplClass) \
  typedef TDocTemplateT<docClass, viewClass> tplClass;           \
  IMPLEMENT_STREAMABLE_FROM_BASE(tplClass, TDocTemplate);        \
  docClass* tplClass::IsMyKindOfDoc(TDocument& doc)              \
  {                                                              \
    return TYPESAFE_DOWNCAST(&doc, docClass);                    \
  }                                                              \
  viewClass* tplClass::IsMyKindOfView(TView& view)               \
  {                                                              \
    return TYPESAFE_DOWNCAST(&view, viewClass);                  \
  }
```

The macro first declares a template class, typdefing it as `tplClass`, then it calls a macro to setup support for persistence, then it defines a couple of member functions. I would have preferred a simpler syntax to create a document template. For example, in MFC all you have to do is instantiate a template class, passing to it a few pointers, with an expression like:

```
CDocTemplate* myTemplate = new CMultiDocTemplate(IDR_MYTYPE,
                                                 RUNTIME_CLASS(CMyDocument),
                                                 RUNTIME_CLASS(CMDIChildWnd),
                                                 RUNTIME_CLASS(CMyView) );
```

The MFC code is more object-oriented, but unfortunately also commits sins of its own, making use of macros again. The OWL code is complicated because it uses a template class to handle Doc/View association. While I'm certainly an advocate of C++ template classes, I'm not sure exactly what benefits there are in using a template class to hold Doc/View associations. MFC uses generic pointers to Document and View objects in the Template class, using polymorphism to call the correct functions at runtime, and this appears to be simple, object-oriented and typesafe.

Document templates facilitate the use of views and documents by handling a number of standard functions. For example when you use commands like **File Open** or **File Save As** on the main menu (assuming your application has these commands), the document template takes over. It presents you with standard dialog boxes, showing the files with a given suffix in a given default directory.

To use a document template, you create a global object like this:

```
TMyTemplate templateObject("View Hex Data",
                           "*.bat","C:\\", 0, dtAutoDelete);
```

The first parameter is a string displayed in a small popup window if you select the **File New** command and your document has multiple viewers. The string describes how the viewer displays its data. If you had a viewer that displayed postscript files graphically, the string might read *View Postscript graphics*.

The other parameters passed to the document template object are the suffix filter used to display files in the File dialog boxes, the default directory to use, the default file suffix to use, and a document template flag value.

A short example

To show how documents, viewers and document templates simplify program development, I'll develop a short browser application, called VIEWER, that lets you view files in two different ways: as text and as hex data. The **File** menu of VIEWER has the standard commands **New**, **Open** and **Close**, all supported automatically by OWL code. The are no **File | Save** or **File | Save As** commands in VIEWER, since the application is not designed to let you make changes to files, but OWL does support these and many other standard commands.

When a document has more than more viewer, a small problem arises when you ask OWL to create a new file of that type, or open a preexisting document. Which viewer should OWL use? Only the user knows, so OWL asks the user. When you select the **File | New** command in VIEWER, OWL displays the small popup menu shown in figure 3.

Figure 3 - The viewer selection popup menu displayed by OWL in VIEWER to select a document viewer.

The text in the popup menu is defined by the programmer, in the document template objects. You select the viewer type desired, then press ENTER, and OWL shows the document accordingly. If you open a document with the **File | Open** command, the common File Open dialog box is displayed, in which the `List Files Of Type` combo box is used to select the desired viewer, as shown in figure 4.

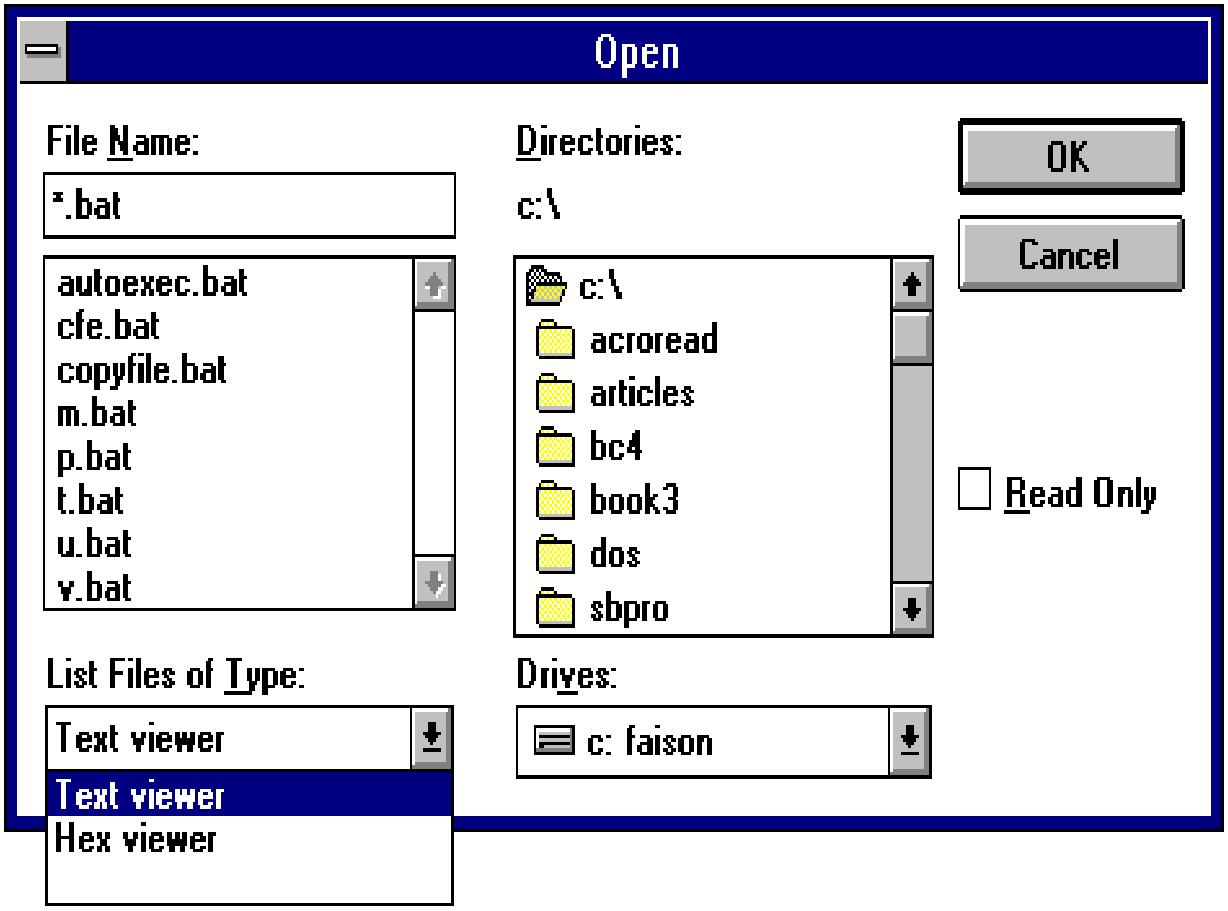

Figure 4 - The viewer selection combo box in the File Open dialog box used by OWL.

As stated earlier, OWL provides two standard viewer classes `TEditView` and `TListView`. The former displays data in text format. It supports standard editing operations, such as Cut, Copy, Paste and Undo, plus text searching commands like Find, Replace and Next.

I used a `TEditView` object to display text in VIEWER. For hex data, I derived a simple class from `TEditView`, and made a few changes. Figure 5 shows VIEWER after opening one file in text mode and one file in hex mode.

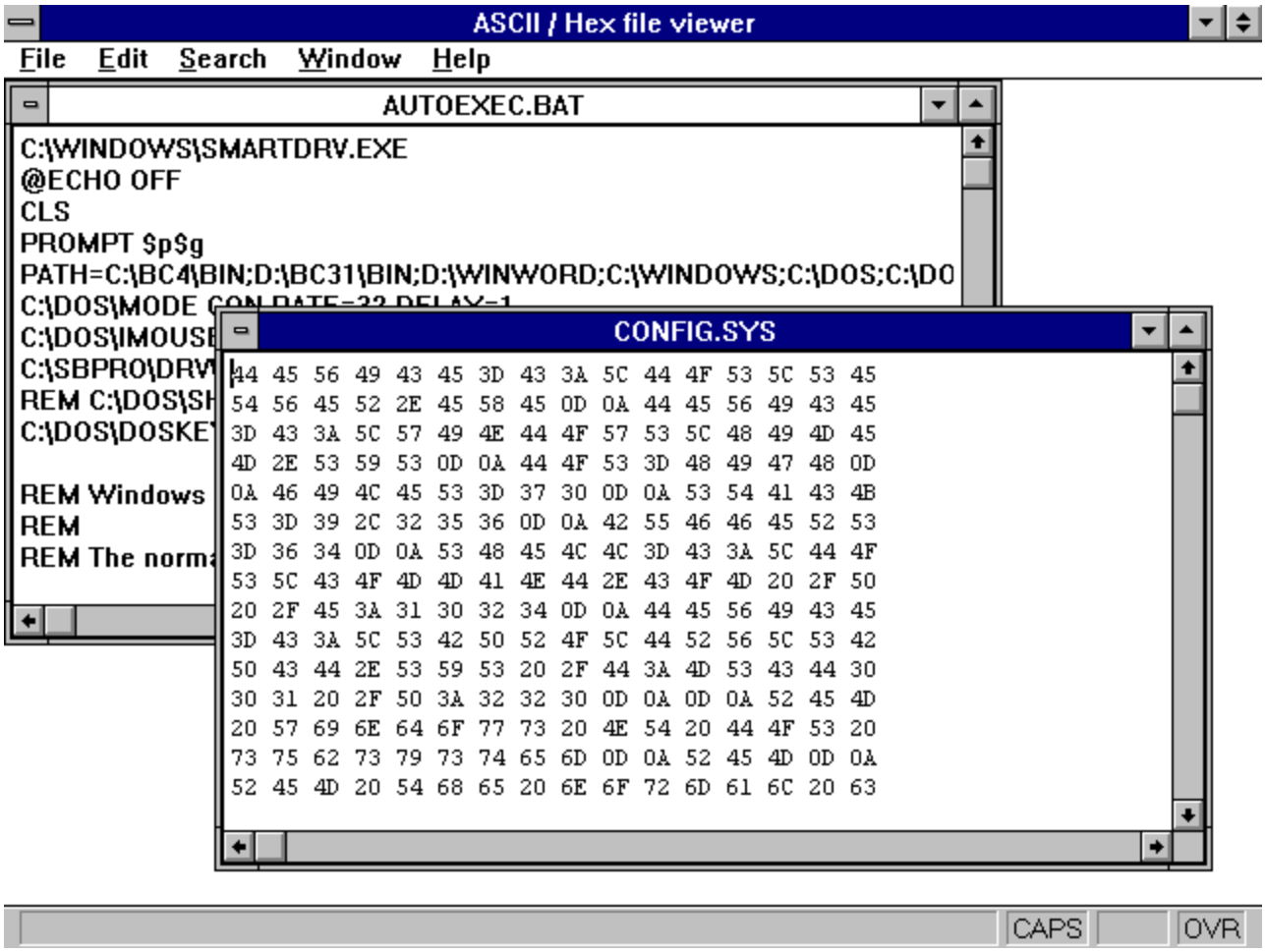

Figure 5 - The two viewers supported by VIEWER, for text and hex data.

I called the hex viewer class `THexView`. I really only added two features to the class, to make it use a fixed pitch font and to display its data in hex/ASCII format. Listing 1 shows the declaration for class `THexView`, and Listing 2 shows the implementation.

```
#ifndef __HEXVIEW_HPP
#define __HEXVIEW_HPP

#include <owl\editview.h>

class THexView : public TEditView {

  char HexToAscii(char);
  HFONT fixedPitchFont;

public:

  THexView(TDocument& doc, TWindow* parent = 0);
  ~THexView();

  static LPCSTR StaticName() {return "Hex View";}
  BOOL Create();
  void Flush() { }
};

#endif
```

Listing 1 - The declaration of the hex viewer class.

```
#include "hexview.hpp"
#include <strstrea.h>

THexView::THexView(TDocument& doc, TWindow* parent)
        : TEditView(doc, parent)
{
  fixedPitchFont = CreateFont(15, 0, 0, 0, 0, 0, 0, 0,
                              0, 0, 0, 0, FIXED_PITCH,
                              "Courier New");
}

THexView::~THexView()
{
  DeleteObject(fixedPitchFont);
}

BOOL THexView::Create(){
  if (!TEditView::Create() )
    return FALSE;

  char* data = LockBuffer();
  UnlockBuffer(data, FALSE);
  if (!data) return FALSE;

  // change the data from ASCII to Hex ASCII,
  // e.g. the byte value 0x24 is converted to 0x32 0x34 0x20,
  // so 0x24 would display as the string "24 "
  ostrstream newData;
  for (int charsOnLine = 0; *data; data++) {
    newData << HexToAscii(*data >> 4)
            << HexToAscii(*data)
            << ' ';

    // show only 16 bytes per line
    charsOnLine += 3;
    if (charsOnLine >= 48) {
      charsOnLine = 0;
      newData << "\r\n";
    }
  }

  newData << ends;

  // now empty the edit control and put the converted
  // data into it
  PostMessage(WM_SETFONT, (WPARAM) fixedPitchFont, 0);
  SetWindowText(newData.str() );
  newData.rdbuf()->freeze(0);
  return TRUE;
}

char THexView::HexToAscii(char c)
{
  static char value [] = {"0123456789ABCDEF"};
  return value [c & 0x0F];
}
```

Listing 2 - The implementation of the hex viewer class.

OWL objects are fully persistent in general, but containers in the so-called BIDS container classlib are not, the rationale being that OWL is a Windows product, while the BIDS classlib is completely general-purpose. It is almost inconceivable to develop a non-trivial OWL application without using the container

classlib classes, so you will generally have to add persistence to the BIDS containers you use. See the section on persistence, later in the article, for a description of how to add persistence to BIDS containers. The containers used in MFC 2.5 are part of the library itself, and have built-in support for persistence, also called *serialization* or *streaming*.

Getting back to the code in listing 2, the The member function `THexView::Create()` is where all the work is done in `THexView`. It calls the base class `TEditView` to fill the client area's edit control with text from a file, then changes the data to hex/ASCII format. The class contructor and destructor create and destroy the fixed pitch font used by the viewer. Listing 3 shows the rest of the implementation of VIEWER.

```
#include <owl\applicat.h>
#include <owl\decmdifr.h>
#include <owl\statusba.h>
#include <owl\editview.h>
#include <owl\filedoc.h>
#include <owl\dialog.h>

#include "hexview.hpp"
#include "viewer.rc"

DEFINE_DOC_TEMPLATE_CLASS(TFileDocument, THexView, HexTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TFileDocument, TEditView, EditTemplate);

EditTemplate a("Text viewer", "*.bat","C:\\", 0,
               dtAutoDelete | dtReadOnly);
HexTemplate  b("Hex viewer", "*.*","C:\\", 0,
               dtAutoDelete | dtReadOnly);

class TViewerApp : public TApplication {

  TMDIClient* Client;

public:

  void InitMainWindow();

  void CmEnableSave(TCommandEnabler& handler)
    { handler.Enable(FALSE); }

  void CmEnableSaveAs(TCommandEnabler& handler)
    { handler.Enable(FALSE); }

  void EvNewView(TView&);
  void EvCloseView(TView&);
  void CmHelpAbout();
  DECLARE_RESPONSE_TABLE(TViewerApp);
};

DEFINE_RESPONSE_TABLE1(TViewerApp, TApplication)
  EV_COMMAND_ENABLE(CM_FILESAVE, CmEnableSave),
  EV_COMMAND_ENABLE(CM_FILESAVEAS, CmEnableSaveAs),
  EV_OWLVIEW(dnCreate, EvNewView),
  EV_OWLVIEW(dnClose, EvCloseView),
  EV_COMMAND(CM_HELPABOUT, CmHelpAbout),
END_RESPONSE_TABLE;

void TViewerApp::InitMainWindow()
{
  DocManager = new TDocManager(dmMDI | dmMenu);
  TDecoratedFrame* frame =
    new TDecoratedMDIFrame("ASCII / Hex file viewer", 0,
                           *(Client=new TMDIClient), TRUE);
  TStatusBar* sb = new TStatusBar(frame, TGadget::Recessed,
```

```
                                    TStatusBar::CapsLock |
                                    TStatusBar::NumLock |
                                    TStatusBar::Overtype);
  frame->Insert(*sb, TDecoratedFrame::Bottom);
  MainWindow = frame;
  MainWindow->SetMenuDescr(TMenuDescr(IDM_MAIN,1,0,1,0,1,1) );
}

void TViewerApp::EvNewView(TView& view)
{
  TMDIChild* child = new TMDIChild(*Client, 0, view.GetWindow() );
  child->SetMenuDescr(TMenuDescr(IDM_EDITVIEW,0,2,0,0,0,0) );
  child->Create();
}

void TViewerApp::EvCloseView(TView& view)
{
  TWindow* client = view.GetWindow();
  if (client) {
    TWindow* child = client->Parent;
    if (child) {
      client->SetParent(0);
      child->ShutDownWindow();
    }
  }
}

void TViewerApp::CmHelpAbout()
{
  TDialog(MainWindow, DIALOG_HELPABOUT).Execute();
}

int OwlMain(int, char**)
{
  return TViewerApp().Run();
}
```

Listing 3 - The implementation of VIEWER, an OWL application to browse files in text or hex/ASCII format.

VIEWER sets up the two document templates of type `EditTemplate` and `HexTemplate`. The main application class `TViewerApp` is derived from the standard OWL class `TApplication`. I only added a couple of member functions to `TViewerApp`, to handle a few basic menu commands. The function `TViewerApp::EvNewView()` changes the main menu when a view window is opened, adding commands to support editing. VIEWER demonstrates a new menu handling mechanism used in OWL 2.0. The mechanism is adapted from the in-place activation menu scheme of OLE 2.0, which allows menus from a container document to be merged at runtime with menus from embedded OLE objects. In OWL, the container and activation objects correspond to the MDI frame window and the MDI child windows. The MDI frame window has its own menu, which is shown when no MDI child windows are open. When you open a child window, OWL merges the frame window's menu with the MDI child window's menu. Each MDI child window can have its own menu. When you close an MDI child, OWL automatically restores the menu. The whole mechanism is controlled through menu objects of class `TMenuDescr`, and allows very fancy runtime menu switching in OWL applications with almost no application code of your own.

The function `TViewerApp::InitMainWindow()` creates two types of windows: a `TDecoratedMDIFrame` and a `TStatusBar`. The former handles all the top level MDI commands, such as **Window | Tile** and **File | Open**, while the latter handles the status bar at the bottom of the main window. I'll describe both classes briefly in the next section.

VIEWER doesn't allow you to save files, so it disables the **File | Save** and **File |Save As** commands, using OWL objects of class `TCommandEnabler`, which are equivalent to the `CCmdUI` objects in MFC 2.5.

Window Types

OWL 1.0 provided support for four basic window types: general-purpose windows (class `TWindow`), dialog boxes (class `TDialog`), MDI frame windows (class `TMDIFrame`) and child control windows (`TControl`). Since the introduction of Windows 3.0 back in 1990, several kinds of standard windows have become very popular, such as status bars, messages bars, tool bars, etc. OWL 1.0 required you to implement your own, using the 4 basic window types `TWindow`, `TDialog`, `TMDIFrame` and `TControl`, but OWL 2.0 has been extended to support the new window types, and has classes to simplify other common programming tasks, such as the use of a dialog box as the main window, the creation of position or size-constrained windows, or the development of custom controls. For example, to create a grid custom control, you would derive a class from `TControl`. Class `TControl` supports either directly or indirectly many of the features of child controls, such as parent notification messages, owner-draw functions, painting and persistence. Figure 6 shows the hierarchy of the OWL 2.0 window classes:

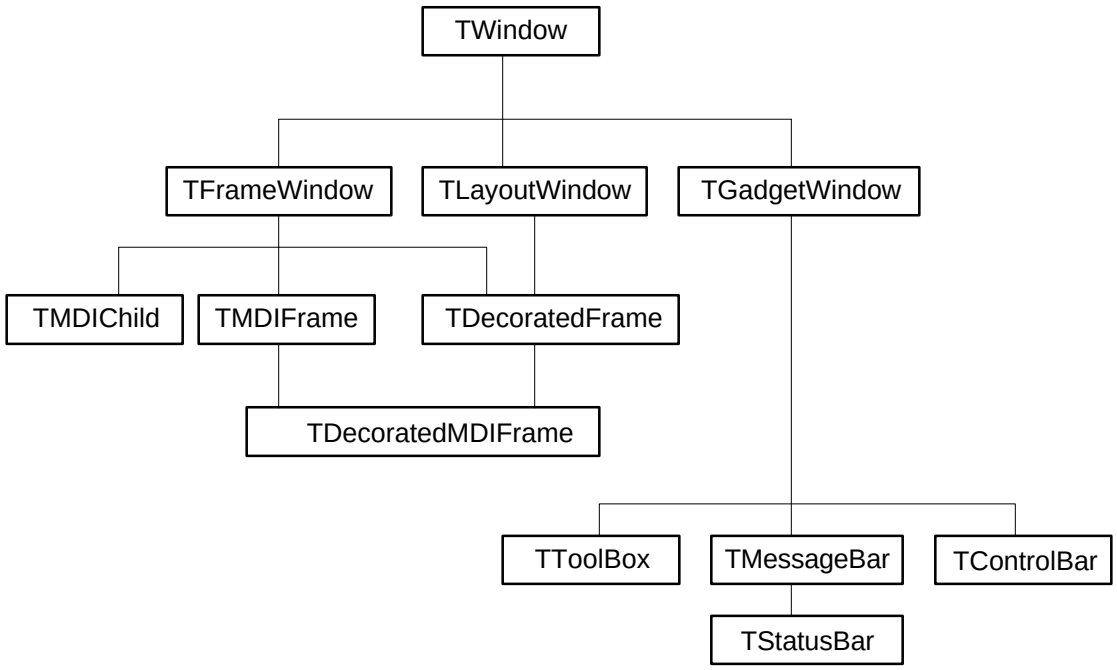

Figure 6 - The hierarchy of basic window types supported by OWL 2.0.

Apart from the basic class `TWindow`, the most important window classes used in typical applications are the following:

- TFrameWindow
- TDecoratedFrame
- TLayoutWindow
- TStatusBar
- TControlBar
- TToolBox

I'll describe the first two one briefly, the latter four a bit more in detail. You use `TFrameWindow` either directly or as a base class when you have a window that also manages a child window in its client area. MDI applications are a good example, because they use a window of class `TMDIClient` to handle the

MDI child windows. OWL applications use a `TFrameWindow` (or derivative) as the main window. Actually, in MDI applications, you use the class `TMDIFrame`, derived from `TFrameWindow`. Class `TFrameWindow` has a parameter allowing it to shrink itself down to the size of the child window, allowing you to make a child control or dialog box look like a regular overlapped window.

You use class `TDecoratedFrame`, and the derived `TDecoratedMDIFrame`, when you have a window to which you want to add status bars, tool bars, or other *decorations*. Class `TDecoratedFrame` has an `insert()` member function, that takes a reference to a window, and a window position. Using `TDecoratedFrame::insert()`, for example, you can add a status bar to the window, and tell `TDecoratedFrame` to put the window at the top, left, right or bottom of the client area. You can add multiple decorations to the same window. `TDecoratedFrame` and `TDecoratedMDIFrame` are the classes used as main windows in the majority of OWL applications.

Layout Windows

Child windows are used for all sorts of things. The only real visual constraint child windows have is that they are clipped to the client area of their parent window, but a child window's size and position are otherwise pretty much independent of the parent. Sure, moving the parent causes the child to also be moved, but that's about it. There is a whole category of child windows whose position, size, or both are required to be a function of some parameter of the parent window. Without searching for exotic examples, just consider a couple a common cases: status bars always need to be at the bottom of the parent, tool bars always at the top. But you might also want to create a child window whose size is a function of the parent's size, or whose height is a function of the width. These types of windows are called *constrained windows*. OWL 2.0 has a special class designed to make it easy to create constrained windows. The class is called `TLayoutWindow`, and lets you set a variety of constraints on the size and position of a window. A short example will help.

Consider an application whose main window has a smaller child window that displays a digital clock. You might want the clock window to always occupy a small portion of the parent, so its left and top borders must be positioned based on the parent's size. You might also want the clock to appear along the lower right border of the parent, so the clock's right and bottom borders will also need to be constrained in terms of the parent's size. Using `TLayoutWindow`, OWL makes creating such a constrained child very easy. All you do is make the parent window a derivative of class `TLayoutWindow`, and use a special layout object of class `TLayoutMetrics` to give the appropriate size and position contraints to the child window. OWL's constrained windows are a rather interesting and unique feature. MFC 2.5 doesn't have any classes to support constrained windows.

To show the details of creating constrained windows, I wrote a short OWL application called LAYOUT that has a main window with a small gray child window positioned near the right bottom border. The size of the child is constrained to be 35% of the parent's size, so resizing the parent you get a differently sized child window. Figures 7 and 8 show how the main and child windows looks for different sizes of the main window.

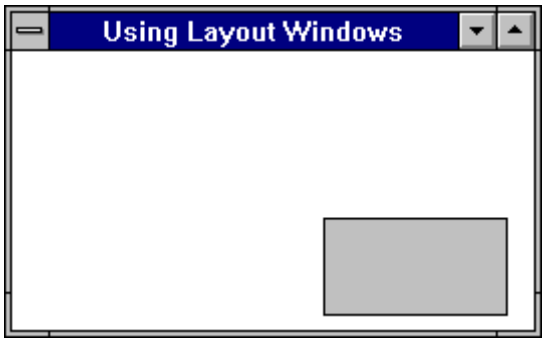

Figure 7 - A child window displayed by LAYOUT.

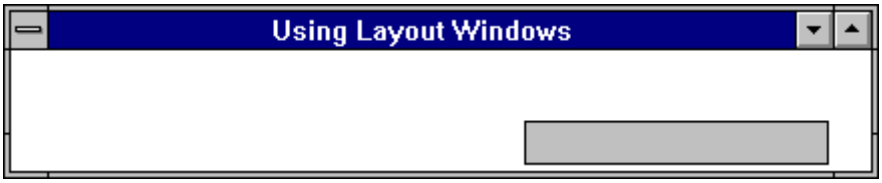

Figure 8 - How the child window is displayed after resizing the parent window in LAYOUT.

The code for LAYOUT is shown in listing 4.

```
#include <owl\framewin.h>
#include <owl\applicat.h>
#include <owl\layoutwi.h>
#include <owl\color.h>

class TClockWindow : public TWindow {
public:
  TClockWindow(TWindow* parent)
        : TWindow(parent, "") {
              SetBkgndColor(TColor(192, 192, 192) );
              Attr.Style = WS_CHILD | WS_BORDER | WS_VISIBLE;
        }
};

class TMyWindow: public TLayoutWindow {

protected:

  TWindow* clock;
  void SetupWindow();

public:

  TMyWindow(TWindow* parent)
        : TLayoutWindow(parent, 0) {
        Attr.Style |= WS_BORDER;
        clock = new TClockWindow(this);
  }
};

void TMyWindow::SetupWindow()
{
  TLayoutWindow::SetupWindow();

  TLayoutMetrics metrics;
  metrics.X.Set(lmLeft, lmPercentOf, lmParent, lmRight, 60);
```

```
  metrics.Y.Set(lmTop, lmPercentOf, lmParent, lmBottom, 60);
  metrics.Width.Set(lmRight, lmPercentOf, lmParent, lmRight, 95);
  metrics.Height.Set(lmBottom, lmPercentOf, lmParent, lmBottom, 95);
  SetChildLayoutMetrics(*clock, metrics);
  Layout();
}

class TLayoutApp : public TApplication {

public:
  void InitMainWindow() {
        MainWindow = new TFrameWindow(0, "Using Layout Windows",
                                      new TMyWindow(0) );
        }
};

int OwlMain(int, char**)
{
  return TLayoutApp().Run();
}
```

Listing 4 - LAYOUT, a short OWL application demonstrating the use of constrained windows with class `TLayoutWindow`.

OWL handles position and size contraints using a set of equations, which are internally solved using matrix inversion. Class `TLayoutMetrics` is used to set the constraints. You can specify the position of any border of a window in terms of either a parameter in the parent window or the child window itself. You can easily create windows that are a percentage of the size of the parent, or that are positioned at a point that depends on the size of the parent. You can also create windows that maintain a constant aspect ratio even after resizing the window. For example, you might want a square child window to have a width of 1/3 the parent's width, thus the height would be constrained to the width. This is easy to accomplish with `TLayoutMetrics`, because you can independently specify a constraint for the x, y, width and height of a window -- as long as one constraint doesn't violate another. LAYOUT uses the constraints:

```
metrics.X.Set(lmLeft, lmPercentOf, lmParent, lmRight, 60);
metrics.Y.Set(lmTop, lmPercentOf, lmParent, lmBottom, 60);
metrics.Width.Set(lmRight, lmPercentOf, lmParent, lmRight, 95);
metrics.Height.Set(lmBottom, lmPercentOf, lmParent, lmBottom, 95);
```

to position and size the child window. The data members `TLayoutMetrics ::X` and `TLayoutMetrics Y` are of class `TEdgeConstraint` . The function `TEdgeConstraint::Set(..)` is declared like this:

```
void  TEdgeConstraint::Set(TEdge edge,
                           TRelationship relationship,
                           TWindow* otherWindow,
                           TEdge otherEdge, int value = 0);
```

The `Width` and `Height` data members are of class `TEdgeOrWidthConstraint`, and have a `Set(...)` member function similar to `TEdgeConstraint`.

Status Bars

Status bars are used in almost every major Windows application today. The information shown on a status bar is highly application-dependent, but many apps use the status line in a dual mode: to display a hint when a menu item is selected, and to display the status of certain keys on the keyboard. For example, selecting the **File** menu in WinWord using the Alt-F key sequence, the string *Creates a new document or template* appears on the status line. Canceling the operation with the ESC key, the status line changes into

a multiple field bar. The fields on the right side show the keyboard options selected. WinWord displays the string OVR when the insert key is in the overtype mode, the string CAPS when the caps lock key is pressed, and so on. OWL handles status bars through the class `TStatusBar`, providing functionality similar to that of WinWord. OWL has built-in support for the keyboard tracking options shown in table 1.

Keyboard Tracking Option	String Displayed
Extended Selection enabled	EXT
Caps Lock enabled	CAPS
Num Lock enabled	NUM
Scroll Lock enabled	SCRL
Overtype enabled	OVR
Macro Recording enabled	REC

Table 1 - The keyboard tracking options supported by OWL.

Objects of class `TStatusBar` don't limit themselves to just tracking the status of the keyboard. You can add other fields to the status bar by using the member function `TStatusBar::insert()` to insert objects derived from class `TGadget`. By doing do, you could easily add a field that displayed the mouse coordinates, a field that showed the row/column position of the insertion bar, the color selected into a painting tool, etc.

Control Bars

Applications like WinWord and BCW use toolbars (called Control Bars in OWL) to expedite operations with a mouse. The tools on a tool bar are the mouse equivalent of accelerator keys for keyboard users. Tool bars have become so widespread that Borland decided to support them directly in OWL, using a class named `TControlBar`. The buttons on the tool bar are of class `TButtonGadget`, and can be positioned freely on the tool bar. Borland provides bitmaps for many common tool bar commands, such as **File New**, **File Open**, **File Print**, **Edit Cut**, **Edit Copy**, etc. To make your own button, all you need to do is provide a bitmap for it (and code to handle it!). For standard operations like **File New** or **File Open**, you often don't need to add any code of your own. Figure 9 shows a sample tool bar created with OWL:

Figure 9 - A simple tool bar created with OWL.

The first two buttons are enabled, the others are grayed out. OWL has code to automatically enable or disable menu and toolbar buttons, based on whether the active window is capable of processing the menu/toolbar button or not. The code to create the tool bar in figure 9 is extremely simple, and shown in listing 5

```
TControlBar* cb = new TControlBar(frame);
cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW));
cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN));
cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE));
cb->Insert(*new TSeparatorGadget(6));
cb->Insert(*new TButtonGadget(CM_EDITCUT, CM_EDITCUT));
cb->Insert(*new TButtonGadget(CM_EDITCOPY, CM_EDITCOPY));
cb->Insert(*new TButtonGadget(CM_EDITPASTE, CM_EDITPASTE));
cb->Insert(*new TSeparatorGadget(6));
cb->Insert(*new TButtonGadget(CM_EDITUNDO, CM_EDITUNDO));
frame->Insert(*cb, TDecoratedFrame::Top);
```

Listing 5 - The code needed to create the toolbar in figure 9.

The code in listing 5 creates the toolbar buttons, using objects of class `TButtonGadget`. The variable `frame` is a pointer to a `TDecoratedFrame` window. The code in listing 5 would typically be part of the `InitMainWindow()` function for the top level application window. Tool bars can be displayed along any border of their parent window.

Tool Boxes

Tool boxes are windows that contain buttons. Tool Boxes may be *floating* or fixed. Floating boxes can be moved around on the screen, fixed ones can't. For example, Resource Workshop uses a floating Tool Box in its Dialog Box editor. Microsoft Draw uses a fixed Tool Box for its drawing tools to create Dialog Boxes. Tool Boxes of both types can easily be created in OWL, using class TToolBox. MFC 2.5 supports fixed Tool Boxes, but not floating ones. Figure 10 shows an example of an OWL Tool Box.

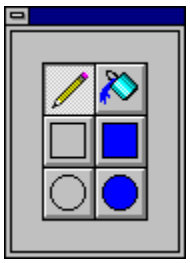

Figure 10 - A sample Tool Box created with OWL.

A floating Tool Box window uses a combination of two control objects for the enclosed buttons: `TToolBox` and `TFloatingPalette`. A `ToolBox` window is the parent window of the buttons. It is a borderless window with a gray background, and lays the child buttons out in a predefined order (by rows or columns), computing the button spacing. A `TFloatingPalette` window uses a `TToolBox` to fill its client area, and handles mouse clicking on the contained buttons.

Exception Handling

Exceptions were recently added to the proposed C++ ANSI draft. OWL 1.0 was developed before exceptions had been accepted into the language, and used an ad hoc error handling mechanism. C++ was inherently weak in dealing with problems arising in object constructors. A very common activity in OWL programs is the creation of window objects. If a window could not be successfully constructed due to insufficient memory, the object constructor had no way to return a value indicating the error.. The function `TModule::MakeWindow()` was used to take a pointer to a `TWindowsObject` object and make sure it was valid. In OWL 1.0, a window was created with code like this:

```
void TMyWindow::CreateWindow()
{
  GetApplication()->MakeWindow(new TNewWindow(this) );
}
```

MakeWindow called `TModule::ValidWindow()` to check the window passed to it. If the window was invalid, due to insufficient memory, an invalid resource name, or other reason, `MakeWindow()` returned a NULL value.

With OWL 2.0, window checking is handled differently, using exceptions. If a window fails to be constructed for any reason, an exception is thrown. Code to create a window is put in a **try** block, resulting in code like this:

```
void TMyWindow::CreateWindow()
{
  try {
    TWindow* window = new TNewWindow(this);
  }
  catch(TException& problem) {
    MessageBox(problem.why().c_str(),
               "Couldn't create a Window");
    throw(problem);  // let OWL abort the application
  }
}
```

The call to `TWindow::MessageBox()` has the parameter

```
problem.why().c_str()
```

which evaluates to a string describing the exception thrown. OWL 2.0 uses a small class hierarchy to handle its own exceptions, as shown in figure 11.

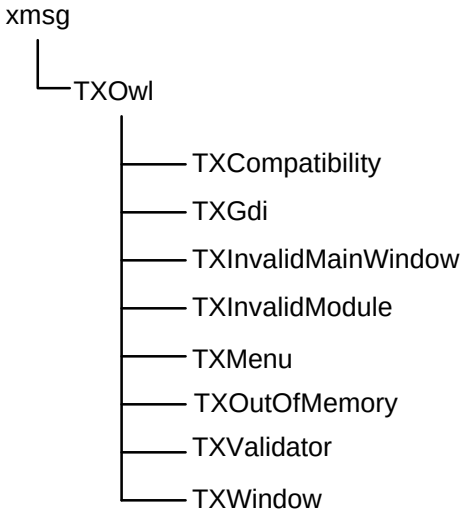

Figure 11 - The OWL class hierarchy dealing with exceptions.

The base class **xmsg** is defined in the ANSI draft proposal. When you create an **xmsg** object, you can pass a **string** to its constructor. **string** is another ANSI defined C++ class, used to manage (what else?) null-terminated character arrays. The function `xmsg::why()` returns a reference to this **string**, which in turn has the function `string::c_str()` which returns a **const char*** pointing at the actual null-terminated array. Class **string** allocates memory for its character array from the global heap, so it doesn't eat up an application's default data segment.

OWL has built-in handlers for the exceptions it throws. These handlers display an error message and typically terminate your application. You can easily change the default error handling by catching OWL exceptions yourself.

Another common exception in OWL programs is **xalloc**. This error is thrown whenever a call to **new** could not be honored due to insufficient or fragmented memory. **xalloc** is another standard C++ exception,

and the default OWL handler displays an error message and terminates your application. OWL 2.0 also accepts the newly adopted *exception interface specifications*, whereby a function can declare the types of exceptions it, or any function called under it, can throw. The use of such specifications requires the system to handle exceptions that were thrown in violation of the declared exception types. Such exceptions are called `unexpected`, and are handled by the function installed by the last call to the standard C++ function `set_unexpected()`. By default the function `unexpected()` is called, which in turn causes your program to be aborted. You can install your own handler for unexpected exceptions, although the default handler will generally be adequate.

Failures in Windows programs can occur when windows are being created, or memory is being allocated. **xalloc** is used when the global **new** operator fails, but memory is also allocated by OWL, with calls to `GlobalAlloc()`, to handle device independent bitmaps inside class `TDib`.. OWL uses the exception `TXGdi` to signal when a `GlobalAlloc()` call fails in an OWL GDI class like `TDib` The string passed in the `TXGdi` exception describes what specific type of problem occurred. OWL doesn't provide a special handler for `TXGdi` exceptions, so unless you have one of your own, the default OWL exception handler will display the usual error message and kill your application.

Templates

Templates were introduced in the ANSI draft for the proposed C++ standard some time ago. OWL 1.0 had already been developed by then, to a good extent, so it made no use of templates. Because of the advantages of class templates in many situations, Borland decided to use templates extensively in OWL 2.0. The following is a list of some of the things templates are used for in OWL 2.0.

- Document templates
- Event Handlers
- DLL Library Manager
- Message Cracking

Templates are also used in the container classes used with OWL 2.0. In the implementation of a container class library, there are really two choices. You can make the containers parameterized, using templates, or you make the containers accept generic items derived from a common base class. For example OWL 1.0 used containers that worked with items derived from the base class `Object`. MFC 2.5 uses containers that take items derived from `CObject`. This approach is very convenient -- especially with heterogeneous containers (containers holding objects of different types), because you can use polymorphism to handle the contained objects.

The drawback of the approach is a certain lake of type safety. If you don't not know a priori the types of the objects in a container, you must use ad hoc identification functions, such as `IsA()` or `IsKindOf()`, or use runtime type identification. The problem is that you need to deal with type identification (or at least typecasting) every time you take an object out of a container, not only forcing the programmer to track object types, but increasing the chances of introducing bugs into the system through invalid type conversions. Templates force you to make a single type commitment at compile time, letting the compiler keep track of types.

The alternative to a template container is a container that takes **void** pointers, which is the approach taken by MFC 2.5. MFC defines containers for certain specific types, such as **unsigned chars**, words and pointers to `CObjects`. Unfortunately, not all data types you create are of this kind. Consider for example a container to handle floating point values. With MFC, you have two choices: you either use a container taking **void*** types, or you create a class, derived from `CObject`, to handle floating point numbers. The first choice requires the use of typecasts on all container operations, which is an un-object-oriented, verbose and bug-prone process. The second choice allows you to use containers that take pointers to

CObjects, which is object-oriented, but unnecessarily complicated since it forces you to create a new class even for scalar items like **float** and **signed char**.

Template containers are a simple solution: for any type of data you can create a specific container type. But what if you need a heterogeneous container, capable of holding different types of objects derived from a common type. For example, you might have two classes `TChair` and `TTable`, both derived from a class `TFurniture`. Using templates, you would create a container taking pointers or references to `TFurniture` objects, using polymorphic calls to handle the objects in the container. Template containers are a natural way to deal with generic types, allowing you to write simpler programs, and simplicity is exactly what object-oriented programming is about.

Message Cracking

OWL 2.0 goes further than 1.0 in encapsulating Windows details. One of the enhancements made is the cracking of Windows messages, a feature also supported by MFC 2.5. Under OWL 1.0, all Windows messages were handled by OWL functions taking a reference to a `TMessage` parameter. `TMessage` was essentially a struct that had a field for an HWND, a WORD, a WPARAM and a LPARAM parameter. Each Windows message handler had to perform explicit type casting on the `TMessage` fields, a process that was entirely un-object-oriented and also error-prone. With message cracking, OWL does all the type conversions on the `TMessage` fields for you, invoking message handlers with the correct types. For example, under OWL 1.0, the WM_CREATE message handler looked like this:

```
void TMyWindow::Create(TMessage& msg)
{
  (CREATESTRUCT FAR*) lpcs = (CREATESTRUCT FAR*) msg.LP;
  // use lpcs...
}
```

With OWL 2.0, the handler looks like this:

```
void TMyWindow::EvCreate(CREATESTRUCT far& lpcs)
{
   // use lpcs...
}
```

All Windows messages are cracked, removing the burden from the programmer, and further reducing the likelihood of bugs getting into the system.

Persistence

One of the greatest stumbling blocks in OWL programs was making an application persistent. Applications with a simple main window were easy to make persistent, but non-trivial applications with hierarchies of multiple windows could be very nasty indeed. Under OWL 1.0, any class that was to support persistence had to be derived from class `TStreamable`, and override 5 member functions, putting each in the right access section. For class `TMyWindow`, the functions would be declared like this in OWL 1.0:

```
public:

    static PTStreamable build();

protected:
```

```
    TMyWindow(StreamableInit);
    virtual void write (opstream& os);
    virtual Pvoid read (ipstream& is);

private:

    virtual const Pchar streamableName() const;
```

where the types `opstream` and `ipstream` are input and output persistent stream objects. OWL 2.0 uses the same basic approach to supporting persistence as OWL 1.0, but introduces a number of simplifications and enhancements. First, you no longer have to add the 5 member functions yourself. The new macro DECLARE_STREAMABLE adds the necessary member functions to a class automatically.

Supporting persistence is not a trivial task for a class library. There are several inherently difficult problems that need to be dealt with, mainly dealing with the issue of pointers. Pointers and streams are incompatible objects, so anytime you insert or extract a pointer from/to a stream there is going to be trouble. During the insertion phase, if a pointer to an object is somehow inserted, then you have to make sure the object pointed at is also inserted. Insertions of multiple pointers to the same object should cause only a single instance of the object referenced to be saved.

During the extraction phase, the problem is even more complex. First of all, all objects must be recognizable in type by the system, so they can be reconstructed and initialized from the streamed data. Second, the pointer problem is still there. Extracting a pointer from a stream requires the object pointed at to also be reconstructed from the stream. If multiple pointers are extracted that reference a common object, the system need to be smart enough to instantiate only a single object, and to set the pointers up correctly. With OWL and other Windows class libraries, there is a further complication represented by Windows hierarchies. Windows have parents, children and siblings. When extracting a window from a stream, OWL must be capable of restoring the hierarchy that was originally inserted. This feat may appear straightforward, but it actually somewhat involved.

OWL takes care of the problems of persistence using an object known as the *stream manager*. This object uses an internal database to keep track of what's going on in the system., When you insert objects into a stream, the manager saves the object information in its database. When inserting pointers, the manager uses the database to avoid inserting multiple instances of the object pointed at. When extracting objects, the stream manager uses the database to determine how to reconstruct new objects, based on type information saved with the object. The database also allows pointers to be resolved correctly.

Although all OWL 2.0 classes are designed to be fully persistent, AppExpert is not capable of creating persistent applications. What is a persistent application ? Say you wanted to create an MDI editor application, similar to the BC 4.0 editor. If you exit BC 4.0 with certain edit windows opened, those same windows will appear the next time you enter BC 4.0. The application knows which edit windows to open because it streamed the windows out when you closed the application, and streamed them back in when you opened the app. Anything you stream in will be in the same state it was when it was streamed out. Persistent applications are very common, and many users have come to expect that a program will start up where it was last left. To create persistent apps with OWL 2.0, you have to add the necessary code yourself, which typically requires cutting and pasting about a page of code from the OWL example programs.

The Visual C++ AppWizard is slightly better, in one sense, because it sets up all skeletal code for a persistent (serializable) application, leaving the developer only with the task of deciding what data to stream in or out, and add a small amount of code. The big problem is that MFC 2.5 doesn't know how to stream entire windows, like MDI child windows, or objects derived from `CWnd`. MFC only lets you stream data objects, such as scalars or objects derived from `CDocument` or `CObject` -- not windows. If you use MFC's document/view model, you can stream the document data in or out, but that isn't sufficient to let an

application startup and restore all the windows that were opened in the previous run, maintaining their old positions, sizes and attributes. With OWL 2.0, this is not only possible, but relatively easy,

OWL containers are not persistent

Borland C++ 4.0 comes with a number of class libraries. One is OWL 2.0, but there are others, like the `iostream` library and the BIDS container library. OWL 1.0 used to have its own containers, which were not only tightly bound to OWL classes, but also persistent in an OWL-compatible way. The OWL 1.0 containers were very simple, and seemed an unnecessary duplication of the code in the full-blown container class library. With OWL 2.0, Borland opted to eliminate the OWL containers completely, and use the general-purpose template BIDS containers.

Consider a window class `TMyWindow` using a template container `TISetAsVector<int>` to hold pointers to integer values. To make the data in the container persistent, in theory you have two alternatives: you either make class `TMyWindow` read and write the container's data, or you derive a class from `TISetAsVector<int>` and make it handle the details of reading and writing its own data. Only the first alternative is supported by OWL. You can't make a template class directly streamable. The macros used to support persistence, such as IMPLEMENT_STREAMABLE1, don't work with template classes. The code in listing 6 shows how you would read and write the container's data from class `TMyWindow`.

```
#include <owl\window.h>
#include <classlib\sets.h>
#include <classlib\objstrm.h>

class TMyWindow : public TWindow {

public:

  // ...

private:

  static void WriteObject(int& obj, void *);
  TISetAsVector<int> Values;

  DECLARE_RESPONSE_TABLE(TMyWindow);
  DECLARE_STREAMABLE(_OWLCLASS, TMyWindow, 1);
};

IMPLEMENT_STREAMABLE1(TMyWindow, TWindow);

void* TMyWindow::Streamer::Read(ipstream& is, uint32) const
{
  ReadBaseObject((TWindow*)GetObject(), is);
  int count;
  is >> count;
  while (count--) {
        int* aNumber;
        is >> *aNumber;
        GetObject()->Values.Add(aNumber);
  }
  return GetObject();
}

void TMyWindow::WriteObject(int& obj, void* os)
{
  *(ofpstream*)os << obj;
}

void TMyWindow::Streamer::Write(opstream& os) const
```

```
{
  WriteBaseObject( (TWindow*)GetObject(), os);
  os << GetObject()->Values.GetItemsInContainer();
  GetObject()->Values.ForEach(WriteObject, &os);
}
```

Listing 6 - Streaming a BIDS container's data from an OWL window.

OWL invokes the functions `Streamer::Read()` and `Streamer::Write()` for each object that participates in a streaming operation. Since the class `Streamer` is a nested class inside each streamable class, to access the outer class' data members requires the use of the function `TStreamer::GetObject()`. In the example in listing 6, the `Write()` function inserts the container's item count into the stream, then each item in the container. The `Read()` function extracts the container's item count from the stream, then each individual item.

Keep track of your children

One of the most common chores for a window is the management of child, sibling and parent windows, which requires the ability to iterate over a collections of windows. Windows objects have their own links to parents, children and siblings, in the form of handles, managed internally by USER. If a window has more than one child, the children are linked together by their sibling handles, and the parent keeps only the handle of the topmost child window. In my discussion here, a child window is not only a window whose WS_CHILD style bit is set, but any window that has a parent. Using this definition, top-level windows are child windows of the Desktop window. Windows uses the window handles to create and manage hierarchies of windows, as shown in figure 12.

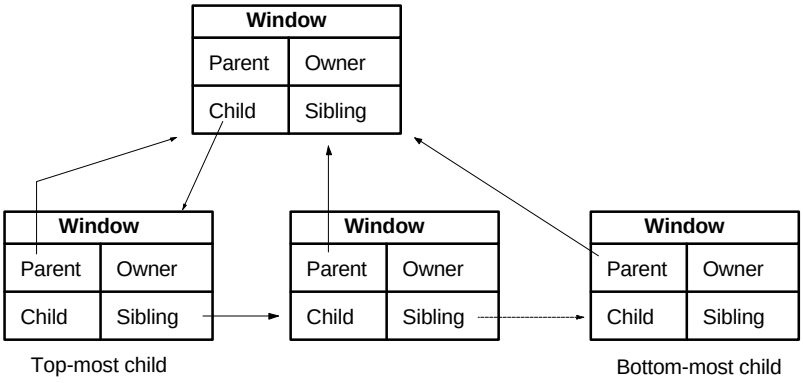

Figure 12 - The window hierarchy maintained by Windows.

OWL uses `TWindow` or derived objects to act as object-oriented stand-ins for windows. Each `TWindow` object has three `TWindow` pointers that together allow OWL to build a hierarchy of `TWindow` objects, parallel to the hierarchy of windows maintained by USER. The `TWindow` pointers were added to make window searching and retrieving faster, and to support a more OOP style in window management. Each OWL window uses the pointers shown in figure 13.

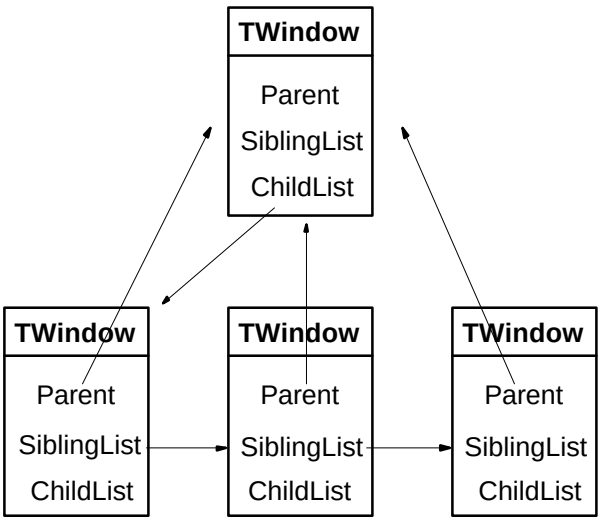

Figure 13 - The used in OWL 2.0 to manage hierarchies of window objects.

OWL doesn't have a TWindow pointer for owner windows, since it never deals with the issue of ownership. OWL has a number of functions that make use of the Parent, ChildList and SiblingList pointers to manage, keep track of, and locate the various windows in a hierarchy, as shown in table 2.

Function Name	Description
ChildWithId	Returns a pointer to the child window with a given ID number
firstThat	iterates over all the child windows, returning a pointer to the first window that satisfies some condition.
forEach	Iterates over all the child windows, performing an operation on each one.
GetFirstChild	Returns a pointer to the first child window of a window, i.e. the first one created.
GetLastChild	Returns a pointer to the last child window or a window.
Next	Returns a pointer to a window's next sibling.
NumChildren	Gives the number of child windows of a window.
Previous	Returns a pointer to a window's previous sibling.

Table 2 - The OWL functions that deal with hierarchical windows.

GDI Support

One of the most visible deficiencies in OWL 1.0 was the lack of support for GDI objects and operations. All drawing on the screen had to be accomplished via direct Windows API calls. OWL 2.0 has been extended to support a number of GDI objects, to make drawing (and printing) easier and more consistent with other OWL constructs. The following list shows the new OWL objects that support GDI operations.

- `TIcon`
- `TCursor`
- `TDib`
- `TDC`

- `TRegion`
- `TBitmap`
- `TFont`
- `TPalette`
- `TBrush`
- `TPen`

Class `TDC` is a base class for a rather extensive hierarchy of derived device context classes, such as `TScreenDC`, `TClientDC`, `TDibDC`, `TMemoryDC`, and so on. Printing and print preview are handled through the built-in OWL classes `TPrintDC` and `TPrinter`. AppExpert generates full support for both print preview and printing. Print preview uses an AppExpert-generated class called `PreviewWindow`, derived from `TDecoratedFrame`, to display a single or double-page preview of a document.

New Controls

Many new Windows programs today make use of controls that weren't available until recently, either because they hadn't been invented yet, or because certain standard ways of doing things hadn't been developed. In the next two sections I'll discuss a couple of new kinds of controls available in OWL that will certainly find use in many applications.

Gadgets

The buttons on tool bars, the text fields in status bars, the bitmapped radio buttons have much in common. These kinds of controls can be handled by traditional means, but they share so many special behaviors and are used so often that Borland decided to develop a new series of classes to handle them, called `TGadget`. Figure 14 shows the hierarchy of OWL gadgets.

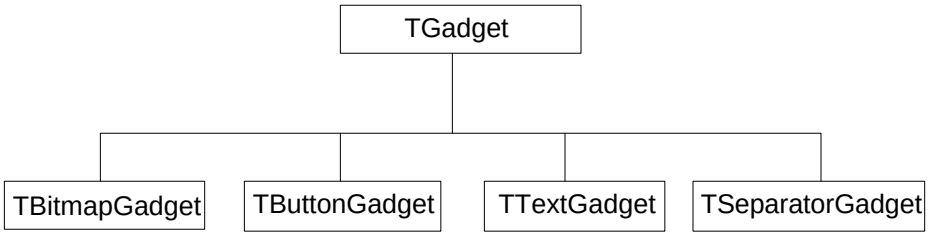

Figure 14 - The hierarchy of control gadgets.

The various `TGadget`-derived classes are basically described by their name. `TBitmapGadget` handles bitmaps, `TButtonGadget` handles buttons, etc. The `TSeparatorGadget` class is used to control the positioning of gadgets in a `TGadgetWindow`. Gadgets are added using the parent window's `insert()` member function, which places the added gadget next to the rightmost gadget on the window. To separate gadgets, you insert a `TSeparatorGadget`, indicating how much separation you want.

`TGadget`-derived classes are not windows, in the ordinary sense. The do occupy space on the screen, and they do interact with the end user, but they are not derived from class `TWindow`, and receive no mouse or keyboard events from Windows. Class `TGadget` is not used directly, but as a base class for other objects to be used in status bars, message bars, tool bars and floating palettes. Only `TGadgetWindows` or derived classes can be used to contain gadget controls, because mouse events and other Windows messages are controlled through the parent `TGadgetWindow`.

Gauges

A gauge is a readout device that represents graphically the value of a variable. Gauges can show values in bar form, numeric form, dial form, or other. Windows programs use gauges typically as progress indicators. Anyone who has installed Windows is familiar with the plain horizontal bar displayed while Setup is copying files.

OWL 2.0 has a new class called `TGauge`, that displays controls that look something like the LED bars used in stereo equipment. You can create either vertical or horizontal gauges. Figure 15 shows a horizontal `TGauge` control.

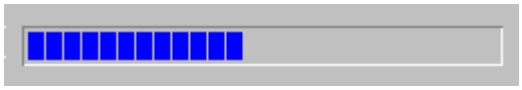

Figure 15 - A horizontal TGauge control.

When you create a `TGauge` object, you indicate its range, which defaults to 0..100, and you set it with calls to the function `TGauge::SetValue()`. TGauge doesn't display its value in numeric form, but a simple `TStatic` control can be used, allowing you to display the text anywhere you want.

Sliders

Sliders are another type of control OWL has borrowed from the world of audio equipment. A slider is a linear control that lets you set the value of a variable. Sliders are frequently used in conjunction with gauges, and can be created in both horizontal and vertical styles. Figure 16 shows a horizontal slider created in OWL 2.0, with the class `THSlider`.

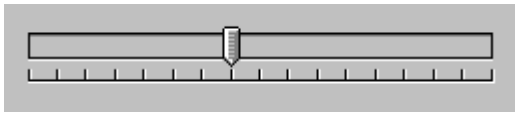

Figure 16 - A horizontal slider control.

Sliders can be adjusted both with the mouse and the keyboard. You read the position of their cursor setting with the function `TSlider::GetPosition()`. The values returned are controlled by the range given to the control with the function `TSlider::SetRange()`, so values are constrained to be between some minimum and maximum value. Most of the attributes of a `TSlider` can be changed, such as the background color, the thickness of the slot area, whether the cursor snaps to the closest tick mark, etc. The tick marks on the slider control are programmable, so you can have as many marks as you want (including none).

Edit Controls with Data Validation

OWL 2.0 doesn't introduce any new classes for text editing, but does enhance the `TEdit` class available in OWL 1.0. Probably the most common thing do you with edit controls is check to see if the user entered something valid -- a process known as *data validation*. If a field requires a value between 30 and 40, then you have to check not only that digits were entered, but also that the number given is within the correct range. If a field is expected to contain a telephone number, then the data entered must also satisfy a certain layout or template. OWL uses so-called *picture* strings to define the layout of data of this type. To have OWL validate an edit field for you, you attach a *validator* object to the edit control. Picture strings and data validator objects are features that are completely missing in MFC 2.5. OWL 2.0 has several types of validators, as shown in figure 17.

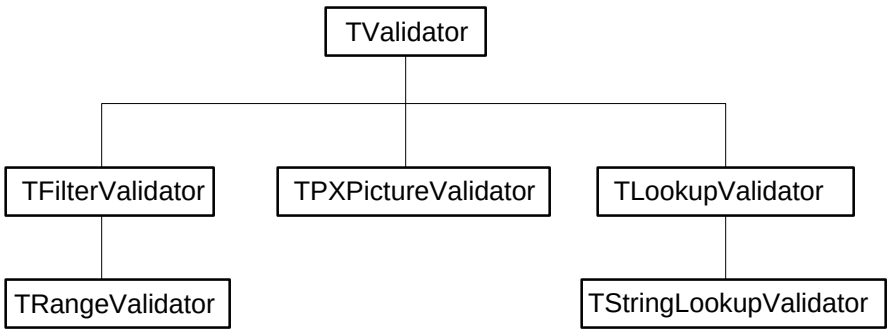

Figure 17 - The edit control data validator class hierarchy in OWL 2.0.

`TFilterValidator` allows you to specify a set of characters that are accepted by an edit control. To create a `TFilterValidator` that only accepts lowercase letters in the range a..g, you can create an object like this:

```
TValidator* validator = new TFilterValidator("abcdefg"));
```

or you can use the shorthand notation:

```
TValidator* validator = new TFilterValidator("a-g"));
```

Once a validator is created, it is attached to an edit control like this:

```
// create an Edit control
TEdit* edit = new TEdit(...);

// create and attach a data validator
edit->SetValidator(new TFilterValidator("a-g") );
```

After attaching a validator, OWL takes full control for checking the contents of an edit control. If you want some special type of validation to take place, all you have to do is derive a class from one of the validator classes and add what you need. When the user types something, attempts to move the focus from a control, or closes a dialog box with the OK button, the validator springs into action. If data is found to be invalid, an error message is displayed.

Getting back to the validator hierarchy in figure 17, a `TPXPictureValidator` allows you to define a pictorial template of the data expected in a control. This template is actually a string, in which certain reserved characters have special meaning. For example, to specify a telephone field, you would use the picture string "###-####". Other characters can be used to accept letters only, digits or letters, letters with case conversion, etc.

Class `TLookupValidator` allows you to provide a list of valid entries. The derived class `TStringLookupValidator` allows you to specify a series of strings. For example, if an edit control accepted only one of the 3 strings Los Angeles, New York, Atlanta, then you could create a validated edit control like this:

```
TStringCollection* cities = new TStringCollection(5);
cities->Add(new string("Los Angeles") );
cities->Add(new string("New York") );
cities->Add(new string("Atlanta") );
TEdit* edit = new TEdit(parentWindow, 101, 10);
edit->SetValidator(new TStringLookupValidator(cities) );
```

You can build custom lookup validators for special cases. For example, you might have a database of valid entries, such as ZIP codes. To create a validator that accesses the database, you would derive a class from `TLookupValidator` and have it search your database for valid entries.

VBX Controls

With the advent of Visual Basic, the market has seen the introduction of an impressive number of VBX custom controls written in that language. OWL treats VBX controls just like native controls like `TEdit` and `TSlider`, so it is very straightforward to use them. There are a couple of extra things you need to do to use VBX controls in your system, to process notification messages from VBX controls.

Since all VBX controls generate notification mesages in a standard way, OWL uses a class called `TVbxEventHandler` to handle these messages. When you create a dialog box that will include VBX controls, you need to multiply derive your dialog box from both the standard `TDialog` class and from `TVbxEventHandler`, like this:

```
class TMyDialog : public TDialog, public TVbxEventHandler {...};
```

You use objects of type `TVbxControl` to encapsulate each of the VBX controls. Next, you need to associate each of the VBX control's notification messages with a dialog box member function. Assuming you had a VBX control that handled spreadsheets, it might have a notification message called "Columns". To associate this message with the member function `TMyDialog::EvColumns()`, you need to put a special EV_VBXEVENTNAME entry in the response table for `TMyDialog`, indicating the ID of the VBX control, like this:

```
DEFINE_RESPONSE_TABLE2(TMyDialog, TDialog, TVbxEventHandler)
  // ...
  EV_VBXEVENTNAME(ID_SPREADSHEET,"Columns",EvColumns),
END_RESPONSE_TABLE;
```

The member function EvColumns must be declared like this:

```
  void EvDropSrc(VBXEVENT far * event);
```

so the dialog box declaration would look something like this:

```
class TMyDialog : public TDialog, public TVbxEventHandler {
public:

  TVbxControl* mySpreadsheet;
  void EvColumns(VBXEVENT far * event);
  DECLARE_RESPONSE_TABLE(TMyDialog);
};
```

Having created an association between the VBX control notification messages and the dialog box's member functions, the rest of the code is fairly straightforward. When the control sends a "Columns" notification, the function `EvColumns()` is invoked by OWL. This function takes `VBXEVENT far*` argument, which you can then use to set or get properties from the VBX control, using code like this:

```
TMyDialog::EvColumns(VBXEVENT far * event)
{
  long property, anotherProperty;

  // read a property
    VBXGetPropByName(event->Control,
                     "SomeProperty", &property);
```

```
  // write a property
    VBXSetPropByName(event->Control,
                     "SomeOtherProperty", anotherProperty);
}
```

OWL provides fairly good encapsulation of the VBX details, yielding an interface that is almost as object-oriented as that of the standard OWL controls. You can use the new Resource Workshop to place VBX controls on your dialog boxes. You can install VBX controls onto the tool palette, and drag and drop the controls just like any other Resource Workshop controls. Although I showed how to manually include a VBX control into a dialog box, ClassExpert can also take care of all the details of generating the correct event response table entries for VBX notification messages, like the Visual C++ ClassWizard. ClassExpert also adds `TVbxEventHandler` as a base class of dialog boxes that incorporate VBX controls. All you have to do is add code inside the body of handlers like `TMyDialog::EvColumns()`, and you're ready to go.

Conclusion

OWL 2.0 is an extensive class library, and to show all of its features would require more space than can be allotted to a single article. I have touched on some of the new features that I thought were of particular interest, but many others remain, such as MDI, Drag and Drop, support of the Windows clipboard, fonts and menus. To summarize, here are some of the features I think are the strong points of OWL 2.0:

- It uses a Document/View architecture
- The class hierarchy has a relatively simple structure, with well thought-out classes
- It makes good use of C++ features such as virtual base classes, multiple inheritance, runtime type identification and polymorphism
- It uses template container classes
- It uses standard C++ exception handling
- It is portable to any C++ compiler compliant with the ANSI draft

OWL is a great library, but it does have a few weaknesses and limitations. Here is a list of the ones I feel are significant:

- no classes to support OLE
- no ODBC support
- no classes to support DDE
- no support for multimedia
- no automatic memory-leak detection

Whether the strengths and limitations I listed will matter to you depend entirely on the kind of applications you develop. If you plan to develop OLE 2.0 clients, containers, or servers, you should seriously consider using MFC 2.5 rather than OWL, given the hideous complexity of OLE programming without an application framework. MFC makes the job extremely simple, and even supports OLE through AppWizard, to generate the type of OLE object you want. As for using DDE in an OWL 2.0 application, the DDEML simplifies the task substantially, making it relatively easy to encapsulate the basics in a couple of small OWL classes. OWL 2.0 doesn't have ODBC support, which is becoming a very hot item in today's world of bound controls and data base accessing. MFC 2.5 does support ODBC, but I wouldn't recommend using MFC only on account of that. ODBC is a pretty simple API that is fairly easy to encapsulate in an OWL class.

One of the best features of OWL 2.0 is its clean object-oriented design which, although not perfect, is well thought out. C programmers may have a steep learning curve with OWL 2.0, but seasoned C++

programmers find OWL relatively easy to work with. A good design makes for a good foundation, allowing subsequent changes and additions to be made without too much trouble. Adding new features to OWL is not only possible but often easy as well.

Sidebar - The first Borland C++ expert: AppExpert

The task of creating a new OWL application with Borland C++ 4.0 is not trivial. Using the old C-style approach, you could cut and paste pieces of applications to build a skeleton framework, then add the functions for you menu commands, create and manage the status line and the tool bar, and so on. Borland decided to take the Microsoft approach, and create a built-in assistant (like the Visual C++ App Wizard), called AppExpert. Although AppExpert is not a part of OWL 2.0 in any way, it dramatically simplifies the task of creating OWL applications. You access AppExpert from the Integrated Development Environment (IDE) of Borland C++ 4.0, through the menu command Project | AppExpert. As with AppWizard, you can only create new applications with AppExpert. Once an application is created, you can no longer use AppExpert to make changes to it. AppExpert presents the following dialog box, through which you can select a vast array of options that affect your application:

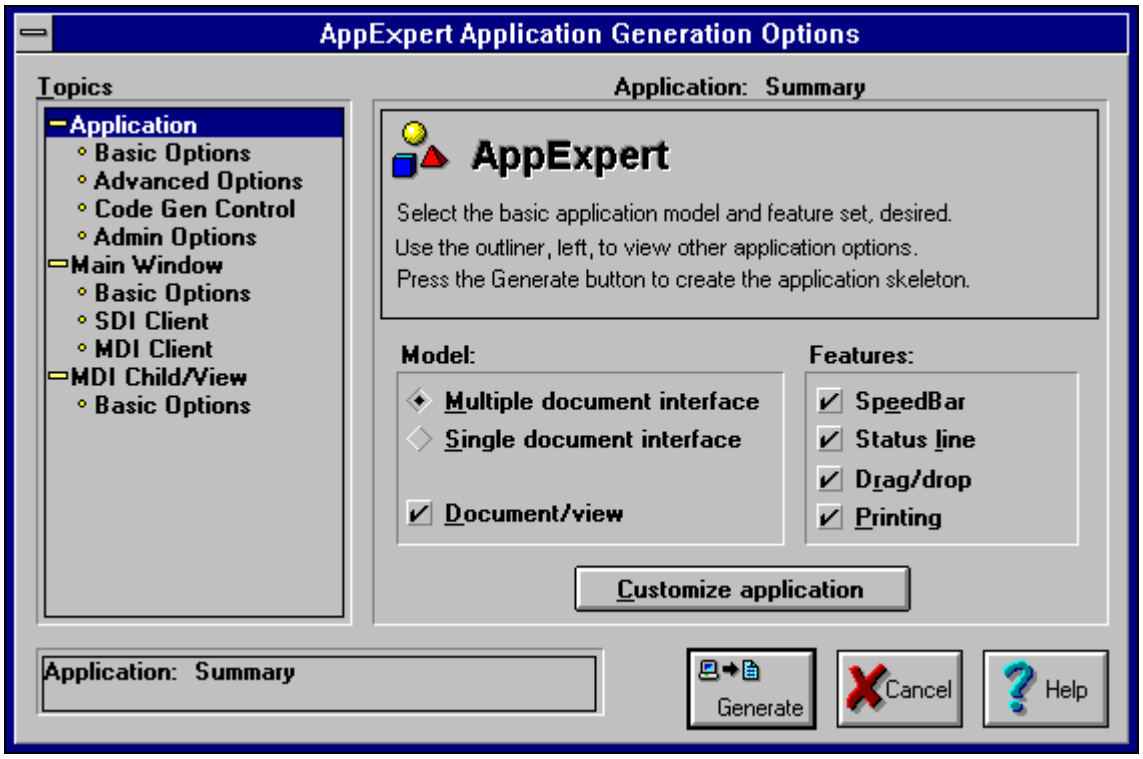

Figure 18 - The AppExpert dialog box.

AppExpert allows you to create both MDI and SDI applications, with the optional use of the Doc/View model. As you can see from figure 18, AppExpert automatically creates code to support the tool bar, the status line, print preview, and drag and drop. In addition, you can control the color and styles of your app's windows, the viewers associated with the child windows (if the Doc/View model was selected), the parameters for the File Open dialog box, the generation of a help file, and the information displayed in the Help About box. Having selected the application features you want, clicking the Generate button will produce all the source files, header files, resource files and support files required for your application. Building the application will require a few minutes of compile and link time, but you will wind up with a completely function -- although skeletal -- application.

Sidebar - The second Borland C++ expert: ClassExpert

AppExpert only knows how to create new OWL applications. To add the functionality required to generate a real application requires you to create additional classes, dialog boxes, resources, menu commands, etc. ClassExpert is designed to assist in writing all the remaining code your application needs, that wasn't already created with AppExpert. As for AppExpert, ClassExpert is not a part of OWL 2.0, but does use and generate code for OWL 2.0. You run ClassExpert by browsing (right-clicking) the project's EXE filename in the Project Manager window, and selecting the command View | ClassExpert. Running ClassExpert gives you a new Smalltalk-like development environment, characterized by a window with three separate panes, as shown in figure 19.

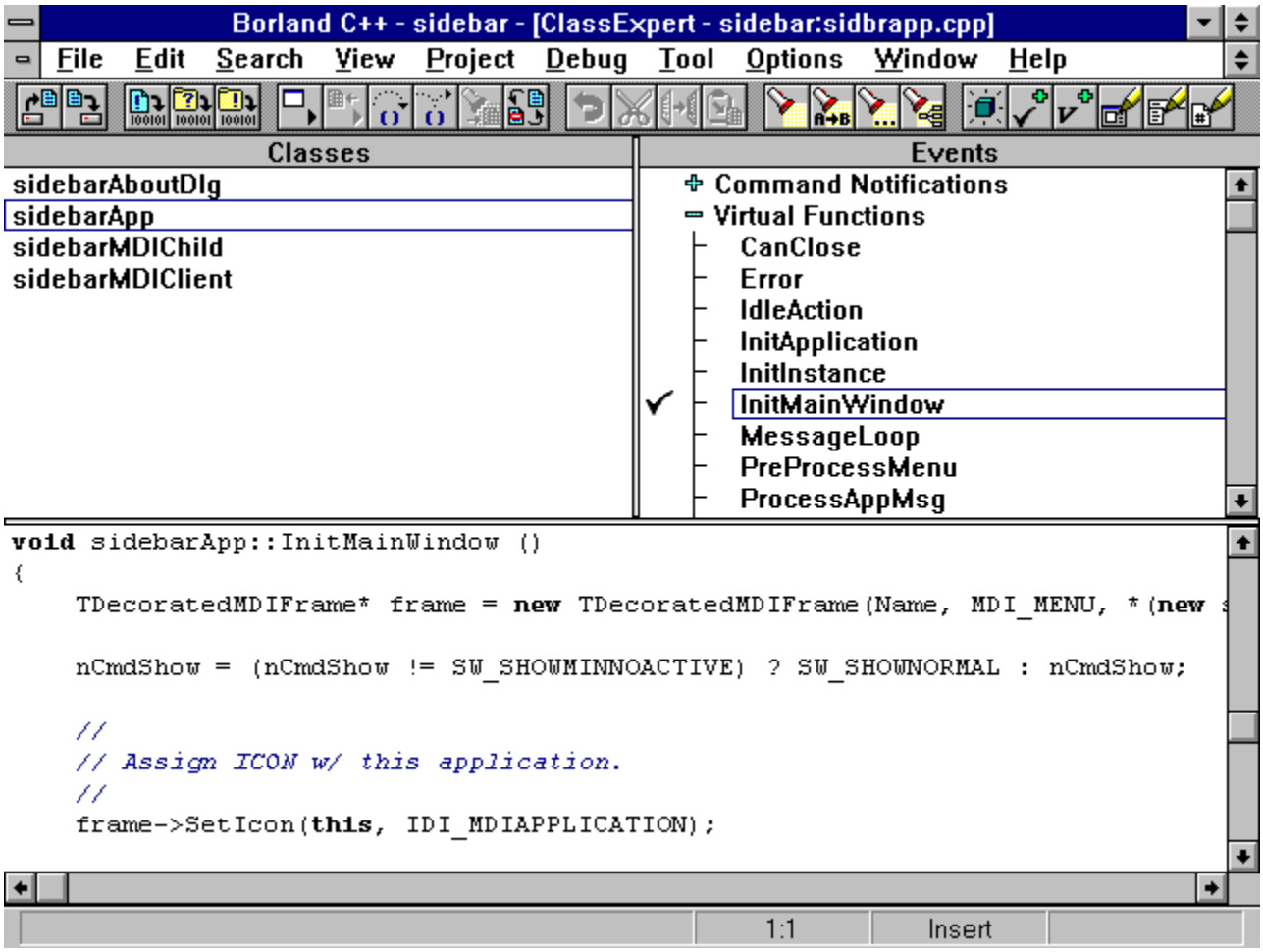

Figure 19 - The ClassExpert development environment.

The top left pane shows a list of the classes in your application. The top right pane shows the Windows messages and handlers for each class. The bottom pane is the source code for the selected class and handler. You can develop all the code for your application using the ClassExpert environment. You can easily jump around in your project by selecting classes and handlers in the ClassExpert panes. To bring up the header file for a class, you just browse (right click) the class name, then select the command **Edit Header**. If you want to change a menu that affects a class, just browse the class name, and select the command **Edit Menu**. Resource Workshop will appear with the menu loaded and ready for editing. To

edit the resource for a dialog box class, browse the class and select **Edit Dialog**, and Resource Workshop will come up on that dialog box.

You can use ClassExpert to create new classes, to add handlers for Windows messages, or to override any of the virtual functions in the base classes. ClassExpert automatically adds code for certain types of functions, and you can add your own code by editing in the lower pane of the ClassExpert window. You can debug your application without ever leaving ClassExpert. By browsing a line of code in the edit pane, you can select the commands `Toggle Breakpoint`, `Run To Cursor`, or `Set Watch`. Borland C++ 4.0 has a class browser, that allows you to display the names of data members and member functions of class objects. The browser can be invoked in ClassExpert by selecting an object in the edit pane, browsing it, and choosing the `Browse Symbol` command. From inside the browser, you can browse individual member functions or data members, or search for the definition and references to a given symbol. The browser is also capable of displaying and printing a graphic representation of your project's entire class hierarchy.