ObjectWindows

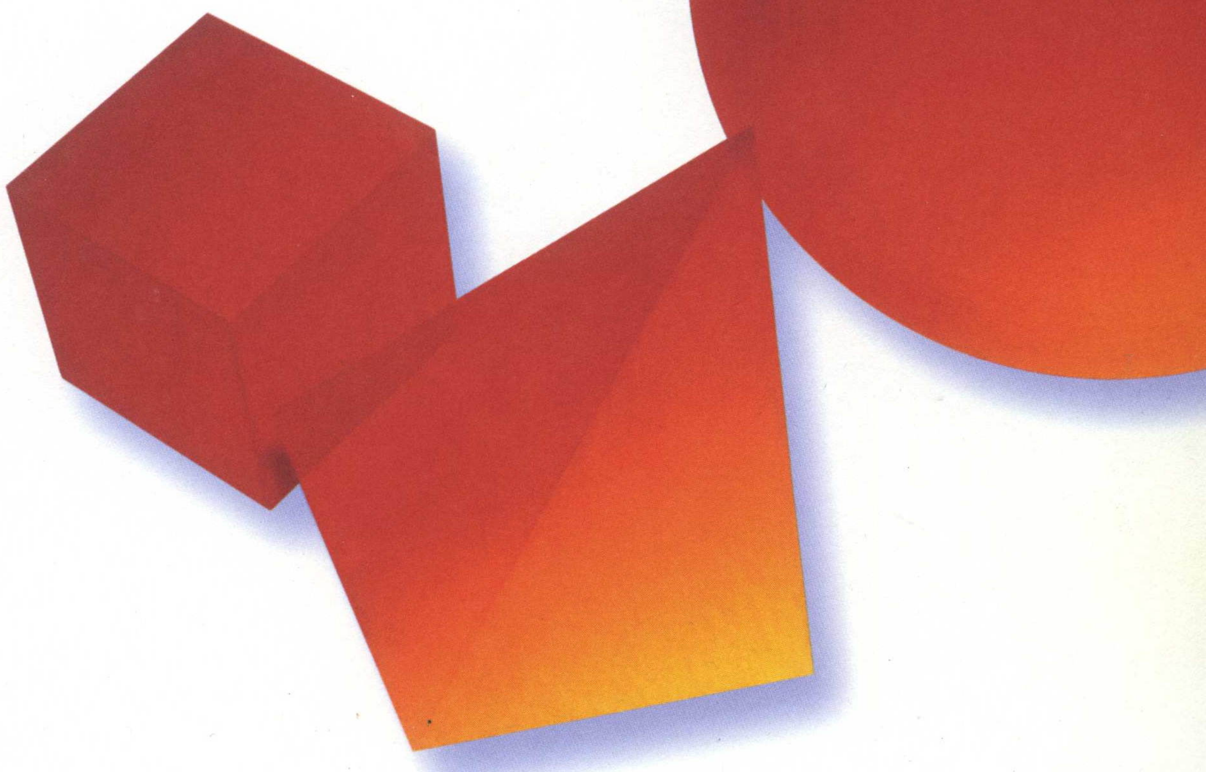
Programmer's Guide

• Advanced Programming
• Command Enabling
• Event Handling
• Using ObjectComponents
• Doc/View Model
• Automation

Borland

# Programmer's Guide

VERSION
2.5



Borland®

# ObjectWindows®

# Borland® ObjectWindows®

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1991, 1994 Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

1E0R1094
9495969798-9 8 7 6 5 4 3 2
H1

Contents


**Introduction** 1

ObjectWindows documentation . . . . . . . . . . 1
Programmer's Guide organization. . . . . . . . . 2
Typefaces and icons used in this book . . . . . . 3

Chapter 1
**Overview of ObjectWindows** 5

Working with class hierarchies . . . . . . . . . . . 5
Using a class . . . . . . . . . . . . . . . . . . . . . 5
Deriving new classes . . . . . . . . . . . . . . . 5
Mixing object behavior . . . . . . . . . . . . . . 6
Instantiating classes . . . . . . . . . . . . . . . . 6
Abstract classes . . . . . . . . . . . . . . . . . . . 7
Inheriting members . . . . . . . . . . . . . . . . 7
Types of member functions . . . . . . . . . . . . 8
Virtual functions . . . . . . . . . . . . . . . . . . 8
Nonvirtual functions . . . . . . . . . . . . . . . 8
Pure virtual functions . . . . . . . . . . . . . . . 9
Default placeholder functions . . . . . . . . . . 9
Object typology . . . . . . . . . . . . . . . . . . . . 9
Window classes . . . . . . . . . . . . . . . . . . . 10
Windows . . . . . . . . . . . . . . . . . . . . . . 10
Frame windows . . . . . . . . . . . . . . . . . . 10
MDI windows . . . . . . . . . . . . . . . . . . . 10
Decorated windows . . . . . . . . . . . . . . . . 10
Dialog box classes . . . . . . . . . . . . . . . . 10
Common dialog boxes . . . . . . . . . . . . . . 10
Other dialog boxes . . . . . . . . . . . . . . . . 11
Control classes . . . . . . . . . . . . . . . . . . 11
Standard Windows controls . . . . . . . . . . 11
Widgets . . . . . . . . . . . . . . . . . . . . . . . 11
Gadgets . . . . . . . . . . . . . . . . . . . . . . . 11
Decorations . . . . . . . . . . . . . . . . . . . . . 11
Graphics classes . . . . . . . . . . . . . . . . . . 12
DC classes . . . . . . . . . . . . . . . . . . . . . 12
GDI objects . . . . . . . . . . . . . . . . . . . . 13
Printing classes . . . . . . . . . . . . . . . . . . 13
Module and application classes . . . . . . . . . 13
Doc/View classes . . . . . . . . . . . . . . . . . 13
Miscellaneous classes . . . . . . . . . . . . . . 14
Menus . . . . . . . . . . . . . . . . . . . . . . . . 14
Clipboard . . . . . . . . . . . . . . . . . . . . . . 14

Chapter 2
**Application and module objects** 15

The minimum requirements . . . . . . . . . . . 16
Including the header file . . . . . . . . . . . . . 16
Creating an object . . . . . . . . . . . . . . . . . 16
Calling the Run function . . . . . . . . . . . . . 16
Finding the object . . . . . . . . . . . . . . . . . 17
Creating the minimum application . . . . . . . 17
Initializing applications . . . . . . . . . . . . . 18
Constructing the application object . . . . . . 18
Using WinMain and OwlMain . . . . . . . . . . 20
Calling initialization functions . . . . . . . . . 21
Initializing the application . . . . . . . . . . . 21
Initializing each new instance . . . . . . . . . 23
Initializing the main window . . . . . . . . . . 23
Specifying the main window display mode . . . . . . . . . . . . . . . . . . . . . . 24
Changing the main window . . . . . . . . . . 25
Application message handling . . . . . . . . . 25
Extra message processing . . . . . . . . . . . . 25
Idle processing . . . . . . . . . . . . . . . . . . . 25
Closing applications . . . . . . . . . . . . . . . . 26
Changing closing behavior . . . . . . . . . . . 26
Closing the application . . . . . . . . . . . . . 26
Modifying CanClose . . . . . . . . . . . . . . . 26
Using control libraries . . . . . . . . . . . . . . 27
Using the Borland Custom Controls Library . . . . . . . . . . . . . . . . . . . . . . . 27
Using the Microsoft 3-D Controls Library . . . 28

Chapter 3
**Interface objects** 29

Why interface objects? . . . . . . . . . . . . . . 30
What do interface objects do? . . . . . . . . . 30
The generic interface object: TWindow . . . . 30
Creating interface objects . . . . . . . . . . . . 31
When is a window handle valid? . . . . . . . . 31
Making interface elements visible . . . . . . . 31
Object properties . . . . . . . . . . . . . . . . . . 32
Window properties . . . . . . . . . . . . . . . . 32
Destroying interface objects . . . . . . . . . . 33
Destroying the interface element . . . . . . . 33
Deleting the interface object . . . . . . . . . . 33
Parent and child interface elements . . . . . . 34
Child-window lists . . . . . . . . . . . . . . . . 34
Constructing child windows . . . . . . . . . . 34
Creating child interface elements . . . . . . . 35
Destroying windows . . . . . . . . . . . . . . . 36
Automatic creation . . . . . . . . . . . . . . . . 37
Manipulating child windows . . . . . . . . . . 37
Operating on all children: ForEach . . . . . . 37
Finding a specific child . . . . . . . . . . . . . 38
Working with the child list . . . . . . . . . . . 38
Registering window classes . . . . . . . . . . . 38

Chapter 4
## Event handling 39


Declaring response tables . . . . . . . . . . . . . . 40
Defining response tables. . . . . . . . . . . . . . . 40
Defining response table entries . . . . . . . . . . . 41
Command message macros . . . . . . . . . . . . 43
Windows message macros . . . . . . . . . . . . 45
Child ID notification message macros . . . . . . 46
EV_CHILD_NOTIFY . . . . . . . . . . . . . . 46
EV_CHILD_NOTIFY_ALL_CODES . . . . . 47
EV_CHILD_NOTIFY_AND_CODE . . . . . 48
EV_CHILD_NOTIFY_AT_CHILD . . . . . . 48


Chapter 5
## Command enabling 49


Handling command-enabling messages . . . . 49
Working with command-enabling objects . . . 51
ObjectWindows command-enabling objects . . 51
TCommandEnabler: The command-enabling interface . . . . . . . . . . . . . . . . . . . . . 51
Functions . . . . . . . . . . . . . . . . . . . . . 52
Data members . . . . . . . . . . . . . . . . . . 53
Common command-enabling tasks . . . . . . . 53
Enabling and disabling command items . . . . 53
Changing menu item text . . . . . . . . . . . . . 55
Toggling command items . . . . . . . . . . . . . 56


Chapter 6
## ObjectWindows exception handling 59


ObjectWindows exception hierarchy . . . . . . 59
Working with TXBase . . . . . . . . . . . . . . . 60
Constructing and destroying TXBase . . . . . 60
Cloning exception objects . . . . . . . . . . . . 60
Throwing TXBase exceptions . . . . . . . . . 61
Working with TXOwl . . . . . . . . . . . . . . . 61
Constructing and destroying TXOwl . . . . . 62
Cloning TXOwl and TXOwl-derived exception objects . . . . . . . . . . . . . . . . 62
Specialized ObjectWindows exception classes . . . . . . . . . . . . . . . . . . . . . . . . 62
ObjectWindows exception-handling macros . . . . . . . . . . . . . . . . . . . . . . . . 63
Turning ObjectWindows exceptions on and off . . . . . . . . . . . . . . . . . . . . . . . 64
Macro expansion . . . . . . . . . . . . . . . . . . 64


Chapter 7
## Window objects 65


Using window objects . . . . . . . . . . . . . . . 65
Constructing window objects . . . . . . . . . . 65
Constructing window objects with virtual bases . . . . . . . . . . . . . . . . . . . 66
Setting creation attributes. . . . . . . . . . . . . 66
Overriding default attributes . . . . . . . . . . 67
Child-window attributes . . . . . . . . . . . . 67
Creating window interface elements . . . . . . 68
Layout windows . . . . . . . . . . . . . . . . . . . 69
Layout constraints . . . . . . . . . . . . . . . . . 70
Defining constraints . . . . . . . . . . . . . . . 70
Defining constraining relationships . . . . . . 73
Indeterminate constraints . . . . . . . . . . . . 74
Using layout windows . . . . . . . . . . . . . . 74
Frame windows . . . . . . . . . . . . . . . . . . . . 75
Constructing frame window objects . . . . . . 76
Constructing a new frame window . . . . . . 76
Constructing a frame window alias . . . . . . 77
Modifying frame windows . . . . . . . . . . . . 78
Decorated frame windows . . . . . . . . . . . . 78
Constructing decorated frame window objects . . . . . . . . . . . . . . . . . . . . . . . 79
Adding decorations to decorated frame windows. . . . . . . . . . . . . . . . . . . . . . . 80
MDI windows . . . . . . . . . . . . . . . . . . . . . 80
MDI applications. . . . . . . . . . . . . . . . . . 81
MDI Window menu . . . . . . . . . . . . . . . 81
MDI child windows . . . . . . . . . . . . . . . 81
MDI in ObjectWindows . . . . . . . . . . . . . 81
Building MDI applications . . . . . . . . . . . . 82
Creating an MDI frame window . . . . . . . . 82
Adding behavior to an MDI client window . . . . . . . . . . . . . . . . . . . . . . 83
Manipulating child windows . . . . . . . . 83
Creating MDI child windows. . . . . . . . . . 83
Automatic child window creation . . . . . 83
Manual child window creation . . . . . . . 84


Chapter 8
## Menu objects 85


Constructing menu objects . . . . . . . . . . . . 85
Modifying menu objects . . . . . . . . . . . . . . 86
Querying menu objects. . . . . . . . . . . . . . . 86
Using system menu objects . . . . . . . . . . . . 87
Using pop-up menu objects . . . . . . . . . . . . 88
Using menu objects with frame windows. . . . 89
Adding menu resources to frame windows . . 89
Using menu descriptors . . . . . . . . . . . . . 90
Creating menu descriptors . . . . . . . . . . . 92
Constructing menu descriptor objects. . . . . 93
Creating menu groups in menu resources . . 93
Merging menus with menu descriptors. . . . 94

## Chapter 9
## Dialog box objects 97


Using dialog box objects . . . . . 97
- Constructing a dialog box object . . . . . 98
  - Calling the constructor . . . . . 98
- Executing a dialog box . . . . . 98
  - Modal dialog boxes . . . . . 98
  - Modeless dialog boxes . . . . . 99
  - Using autocreation with dialog boxes . . . . . 100
  - Managing dialog boxes . . . . . 101
  - Handling errors executing dialog boxes . . . 101
- Closing the dialog box . . . . . 102

Using a dialog box as your main window . . 102

Manipulating controls in dialog boxes . . . . 103
- Communicating with controls . . . . . 104

Associating interface objects with controls . . . . . 104
- Control objects . . . . . 105
- Setting up controls . . . . . 106

Using dialog boxes . . . . . 106
- Using input dialog boxes . . . . . 107
- Using common dialog boxes . . . . . 107
  - Constructing common dialog boxes . . . . . 107
  - Executing common dialog boxes . . . . . 108
- Using color common dialog boxes . . . . . 109
- Using font common dialog boxes . . . . . 110
- Using file open common dialog boxes . . . . . 111
- Using file save common dialog boxes . . . . . 112
- Using find and replace common dialog boxes . . . . . 113
  - Constructing and creating find and replace common dialog boxes . . . . . 113
  - Processing find-and-replace messages . . . . 113
  - Handling a Find Next command . . . . . 114
- Using printer common dialog boxes . . . . . 115


## Chapter 10
## Doc/View objects 117


How documents and views work together . . . . . 117
- Documents . . . . . 119
- Views . . . . . 119
- Associating document and view classes . . . 120
- Managing Doc/View . . . . . 120

Document templates . . . . . 121
- Designing document template classes . . . . . 121
- Creating document registration tables . . . . 122
- Creating template class instances . . . . . 123
- Modifying existing templates . . . . . 125

Using the document manager . . . . . 125
- Constructing the document manager . . . . . 126
- TDocManager event handling . . . . . 127

Creating a document class . . . . . 128
- Constructing TDocument . . . . . 129
- Adding functionality to documents . . . . . 129
- Data access functions . . . . . 129
  - Stream access . . . . . 130
  - Stream list . . . . . 130
  - Complex data access . . . . . 130
  - Data access helper functions . . . . . 131
- Closing a document . . . . . 131
- Expanding document functionality . . . . . 132
- Working with the document manager . . . . 132
- Working with views . . . . . 132

Creating a view class . . . . . 134
- Constructing TView . . . . . 134
- Adding functionality to views . . . . . 134
  - TView virtual functions . . . . . 135
  - Adding a menu . . . . . 135
- Adding a display to a view . . . . . 135
  - Adding pointers to interface objects . . . . . 135
  - Mixing TView with interface objects . . . . 136
- Closing a view . . . . . 136

Doc/View event handling . . . . . 136
- Doc/View event handling in the application object . . . . . 137
- Doc/View event handling in a view . . . . . 138
  - Handling predefined Doc/View events . . . 138
  - Adding custom view events . . . . . 138

Doc/View properties . . . . . 139
- Property values and names . . . . . 140
- Accessing property information . . . . . 141
  - Getting and setting properties . . . . . 141


## Chapter 11
## Control objects 143


Control classes . . . . . 143
- What are control objects? . . . . . 144

Constructing and destroying control objects . . . . . 144
- Constructing control objects . . . . . 144
  - Adding the control object pointer data member . . . . . 145
  - Calling control object constructors . . . . . 145
  - Changing control attributes . . . . . 146
  - Initializing the control . . . . . 146
- Showing controls . . . . . 146
- Destroying the control . . . . . 146

Communicating with control objects . . . . . 147
- Manipulating controls . . . . . 147
- Responding to controls . . . . . 147
- Making a window act like a dialog box . . . 147

Using particular controls . . . . . 147
- Using list box controls . . . . . 148
  - Constructing list box objects . . . . . 148

Modifying list boxes . . . . . . . . . . . . . . . . 148
Querying list boxes . . . . . . . . . . . . . . . . 149
Responding to list boxes . . . . . . . . . . . . 149
Using static controls . . . . . . . . . . . . . . . 150
Constructing static control objects . . . . . . . 150
Modifying static controls . . . . . . . . . . . . 151
Querying static controls . . . . . . . . . . . . . 151
Using button controls . . . . . . . . . . . . . . 151
Constructing buttons . . . . . . . . . . . . . . . 151
Responding to buttons . . . . . . . . . . . . . . 152
Using check box and radio button controls . . . . . . . . . . . . . . . . . . . . . . 152
Constructing check boxes and radio buttons . . . . . . . . . . . . . . . . . . . . . . 153
Modifying selection boxes . . . . . . . . . . . 153
Querying selection boxes . . . . . . . . . . . . 154
Using group boxes . . . . . . . . . . . . . . . . 154
Constructing group boxes . . . . . . . . . . . 154
Grouping controls . . . . . . . . . . . . . . . . 154
Responding to group boxes . . . . . . . . . . 155
Using scroll bars . . . . . . . . . . . . . . . . . 155
Constructing scroll bars . . . . . . . . . . . . . 155
Controlling the scroll bar range . . . . . . . . 155
Controlling scroll amounts . . . . . . . . . . . 156
Querying scroll bars . . . . . . . . . . . . . . . 156
Modifying scroll bars . . . . . . . . . . . . . . . 156
Responding to scroll-bar messages . . . . . . 156
Avoiding thumb tracking messages . . . 157
Specializing scroll bar behavior . . . . . . 157
Using sliders and gauges . . . . . . . . . . . . 158
Using edit controls . . . . . . . . . . . . . . . . 159
Constructing edit controls . . . . . . . . . . . 159
Using the Clipboard and the Edit menu . . . 160
Querying edit controls . . . . . . . . . . . . . . 160
Modifying edit controls . . . . . . . . . . . . . 161
Using combo boxes . . . . . . . . . . . . . . . . 162
Varieties of combo boxes . . . . . . . . . . . . 162
Choosing combo box types . . . . . . . . . . . 163
Constructing combo boxes . . . . . . . . . . . 163
Modifying combo boxes . . . . . . . . . . . . . 164
Querying combo boxes . . . . . . . . . . . . . . 164
Setting and reading control values . . . . . . . 164
Using transfer buffers . . . . . . . . . . . . . . 164
Defining the transfer buffer . . . . . . . . . . . 165
List box transfer . . . . . . . . . . . . . . . . . . 166
Combo box transfer . . . . . . . . . . . . . . . . 167
Defining the corresponding window or dialog box . . . . . . . . . . . . . . . . . . . . 167
Using transfer with a dialog box . . . . . . . 168
Using transfer with a window . . . . . . . . . 168
Transferring the data . . . . . . . . . . . . . . . 168
Transferring data to a window . . . . . . . . . 168
Transferring data from a dialog box . . . . . 169
Transferring data from a window . . . . . . . 169
Supporting transfer for customized controls . . . . . . . . . . . . . . . . . . . . . 169

Chapter 12
**Gadget and gadget window objects 171**

Gadgets . . . . . . . . . . . . . . . . . . . . . . . . 171
Class TGadget . . . . . . . . . . . . . . . . . . . 171
Constructing and destroying TGadget . . . . 171
Identifying a gadget . . . . . . . . . . . . . . . 172
Modifying and accessing gadget appearance . . . . . . . . . . . . . . . . . . . . 173
Bounding the gadget . . . . . . . . . . . . . . . 173
Shrink wrapping a gadget . . . . . . . . . . . . 173
Setting gadget size . . . . . . . . . . . . . . . . 174
Matching gadget colors to system colors . . . 174
TGadget public data members . . . . . . . . . 174
Enabling and disabling a gadget . . . . . . . . 174
Deriving from TGadget . . . . . . . . . . . . . . 175
Initializing and cleaning up . . . . . . . . . . . 175
Painting the gadget . . . . . . . . . . . . . . . . 175
Invalidating and updating the gadget . . . . 176
Mouse events in a gadget . . . . . . . . . . . . 176
ObjectWindows gadget classes . . . . . . . . . 177
Class TSeparatorGadget . . . . . . . . . . . . . 178
Class TTextGadget . . . . . . . . . . . . . . . . . 178
Constructing and destroying TTextGadget . . . . . . . . . . . . . . . . . . . . . . 178
Accessing the gadget text . . . . . . . . . . . . 178
Class TBitmapGadget . . . . . . . . . . . . . . . 179
Constructing and destroying TBitmapGadget . . . . . . . . . . . . . . . . . . . . 179
Selecting a new image . . . . . . . . . . . . . . 179
Setting the system colors . . . . . . . . . . . . 179
Class TButtonGadget . . . . . . . . . . . . . . . 180
Constructing and destroying TButtonGadget . . . . . . . . . . . . . . . . . . . . 180
Accessing button gadget information . . . . 181
Setting button gadget style . . . . . . . . . . . 181
Command enabling . . . . . . . . . . . . . . . . 182
Setting the system colors . . . . . . . . . . . . 182
Class TControlGadget . . . . . . . . . . . . . . . 182
Constructing and destroying TControlGadget . . . . . . . . . . . . . . . . . . . . 182
Gadget windows . . . . . . . . . . . . . . . . . . 182
Constructing and destroying TGadgetWindow . . . . . . . . . . . . . . . . . . . . 183
Creating a gadget window . . . . . . . . . . . 183
Inserting a gadget into a gadget window . . . 183
Removing a gadget from a gadget window . . . . . . . . . . . . . . . . . . . . . . . 184
Setting window margins and layout direction . . . . . . . . . . . . . . . . . . . . . 184
Laying out the gadgets . . . . . . . . . . . . . . 184
Notifying the window when a gadget changes size . . . . . . . . . . . . . . . . . . . 185

Shrink wrapping a gadget window . . . . . . 185
Accessing window font . . . . . . . . . . . . . . 186
Capturing the mouse for a gadget. . . . . . . 186
Setting the hint mode . . . . . . . . . . . . . . 186
Idle action processing . . . . . . . . . . . . . . 187
Searching through the gadgets. . . . . . . . . 187
Deriving from TGadgetWindow . . . . . . . . 187
Painting a gadget window . . . . . . . . . . . 187
Size and inner rectangle. . . . . . . . . . . . . 188
Layout units. . . . . . . . . . . . . . . . . . . . . 188
Message response functions . . . . . . . . . . 189
ObjectWindows gadget window classes . . . 189
Class TControlBar . . . . . . . . . . . . . . . . . 189
Class TMessageBar . . . . . . . . . . . . . . . . . 190
Constructing and destroying TMessageBar . . . . . . . . . . . . . . . . . . 190
Setting message bar text. . . . . . . . . . . . . 190
Setting the hint text . . . . . . . . . . . . . . . 190
Class TStatusBar . . . . . . . . . . . . . . . . . 191
Constructing and destroying TStatusBar. . . 191
Inserting gadgets into a status bar. . . . . . . 191
Displaying mode indicators . . . . . . . . . . 191
Spacing status bar gadgets . . . . . . . . . . . 192
Class TToolBox . . . . . . . . . . . . . . . . . . . 192
Constructing and destroying TToolBox . . . 192
Changing tool box dimensions. . . . . . . . . 193

## Chapter 13
## **Printer objects** 195

Creating a printer object . . . . . . . . . . . . . 195
Creating a printout object . . . . . . . . . . . . 197
Printing window contents. . . . . . . . . . . . . 198
Printing a document . . . . . . . . . . . . . . . 199
Setting print parameters. . . . . . . . . . . . . 199
Counting pages . . . . . . . . . . . . . . . . . . 199
Printing each page . . . . . . . . . . . . . . . . 200
Indicating further pages . . . . . . . . . . . . . 200
Other printout considerations . . . . . . . . . 200
Choosing a different printer. . . . . . . . . . . 201

## Chapter 14
## **Graphics objects** 203

GDI class organization. . . . . . . . . . . . . . 203
Changes to encapsulated GDI functions . . . 204
Working with device contexts . . . . . . . . . 206
TDC class . . . . . . . . . . . . . . . . . . . . . 206
Constructing and destroying TDC . . . . . . 207
Device-context operators . . . . . . . . . . . . 207
Device-context functions . . . . . . . . . . . . 208
Selecting and restoring GDI objects . . . . . . 208
Drawing tool functions . . . . . . . . . . . . . . 209
Color and palette functions. . . . . . . . . . . 209
Drawing attribute functions . . . . . . . . . . 209
Viewport and window mapping functions . . . . . . . . . . . . . . . . . . . . 210
Coordinate functions. . . . . . . . . . . . . . . 210
Clip and update rectangle and region functions . . . . . . . . . . . . . . . . . . . . 210
Metafile functions . . . . . . . . . . . . . . . . 210
Current position functions . . . . . . . . . . . 210
Font functions. . . . . . . . . . . . . . . . . . . 211
Path functions. . . . . . . . . . . . . . . . . . . 211
Output functions . . . . . . . . . . . . . . . . . 211
Object data members and functions . . . . . 213
TPen class. . . . . . . . . . . . . . . . . . . . . . 213
Constructing TPen. . . . . . . . . . . . . . . . . 213
Accessing TPen. . . . . . . . . . . . . . . . . . . 215
TBrush class . . . . . . . . . . . . . . . . . . . . 215
Constructing TBrush . . . . . . . . . . . . . . . 215
Accessing TBrush . . . . . . . . . . . . . . . . . 216
TFont class . . . . . . . . . . . . . . . . . . . . . 217
Constructing TFont . . . . . . . . . . . . . . . . 217
Accessing TFont . . . . . . . . . . . . . . . . . . 218
TPalette class. . . . . . . . . . . . . . . . . . . . 218
Constructing TPalette . . . . . . . . . . . . . . . 218
Accessing TPalette. . . . . . . . . . . . . . . . . 219
Member functions . . . . . . . . . . . . . . . . 219
Extending TPalette. . . . . . . . . . . . . . . . . 220
TBitmap class . . . . . . . . . . . . . . . . . . . 221
Constructing TBitmap. . . . . . . . . . . . . . . 221
Accessing TBitmap . . . . . . . . . . . . . . . . 222
Member functions . . . . . . . . . . . . . . . . 223
Extending TBitmap . . . . . . . . . . . . . . . . 224
TRegion class . . . . . . . . . . . . . . . . . . . 224
Constructing and destroying TRegion . . . . 224
Accessing TRegion. . . . . . . . . . . . . . . . . 226
Member functions . . . . . . . . . . . . . . . . 226
Operators . . . . . . . . . . . . . . . . . . . . . 227
TIcon class . . . . . . . . . . . . . . . . . . . . . 229
Constructing TIcon . . . . . . . . . . . . . . . . 229
Accessing TIcon . . . . . . . . . . . . . . . . . . 230
TCursor class. . . . . . . . . . . . . . . . . . . . 231
Constructing TCursor. . . . . . . . . . . . . . . 231
Accessing TCursor. . . . . . . . . . . . . . . . . 232
TDib class. . . . . . . . . . . . . . . . . . . . . . 232
Constructing and destroying TDib . . . . . . 233
Accessing TDib. . . . . . . . . . . . . . . . . . . 234
Type conversions. . . . . . . . . . . . . . . . . 234
Accessing internal structures . . . . . . . . . 234
DIB information . . . . . . . . . . . . . . . . . 235
Working in palette or RGB mode . . . . . . 235
Matching interface colors to system colors . . . . . . . . . . . . . . . . . . . . . . . 237
Extending TDib . . . . . . . . . . . . . . . . . . 237

Chapter 15
## Validator objects 239

The standard validator classes . . . . . . . . . . 239
Validator base class. . . . . . . . . . . . . . . . . 240
Filter validator class . . . . . . . . . . . . . . . . 240
Range validator class. . . . . . . . . . . . . . . . 240
Lookup validator class. . . . . . . . . . . . . . . 240
String lookup validator class . . . . . . . . . . 241
Picture validator class . . . . . . . . . . . . . . . 241
Using data validators. . . . . . . . . . . . . . . . 242
Constructing an edit control object. . . . . . . 242
Constructing and assigning validator objects. . . . . . . . . . . . . . . . . . . . . . . 242
Overriding validator member functions . . . 243
Member function Valid . . . . . . . . . . . . . 243
Member function IsValid . . . . . . . . . . . . 243
Member function IsValidInput. . . . . . . . . 243
Member function Error . . . . . . . . . . . . . 244

Chapter 16
## Visual Basic controls 245

Using VBX controls. . . . . . . . . . . . . . . . . 245
VBX control classes. . . . . . . . . . . . . . . . . 246
TVbxControl class . . . . . . . . . . . . . . . . 246
TVbxControl constructors . . . . . . . . . . . 247
Implicit and explicit construction . . . . . . . 248
TVbxEventHandler class . . . . . . . . . . . . 249
Handling VBX control messages. . . . . . . . 249
Event response table . . . . . . . . . . . . . . . 249
Interpreting a control event. . . . . . . . . . . 250
Finding event information . . . . . . . . . . . 251
Accessing a VBX control. . . . . . . . . . . . . 251
VBX control properties. . . . . . . . . . . . . . 251
Finding property information . . . . . . . . . 252
Getting control properties . . . . . . . . . . . 252
Setting control properties. . . . . . . . . . . . 253
VBX control methods . . . . . . . . . . . . . . 253

Chapter 17
## ObjectWindows dynamic-link libraries 255

Writing DLL functions. . . . . . . . . . . . . . . 255
DLL entry and exit functions . . . . . . . . . . 256
LibMain . . . . . . . . . . . . . . . . . . . . . . . 256
WEP . . . . . . . . . . . . . . . . . . . . . . . . . 256
DllEntryPoint. . . . . . . . . . . . . . . . . . . . 257
Exporting DLL functions . . . . . . . . . . . . 257
Importing (calling) DLL functions . . . . . . . 257
Writing shared ObjectWindows classes. . . . 258
Defining shared classes . . . . . . . . . . . . . 258
The TModule object. . . . . . . . . . . . . . . . 259
Using ObjectWindows as a DLL . . . . . . . . 260
Calling an ObjectWindows DLL from a non-ObjectWindows application . . . . . . . 260
Implicit and explicit loading. . . . . . . . . . . 261
Mixing static and dynamic-linked libraries. . 261

Chapter 18
## Support for OLE in Borland C++ 263

What does ObjectComponents do? . . . . . . 263
Where should you start? . . . . . . . . . . . . . 264
Writing applications. . . . . . . . . . . . . . . . 264
Creating a new application . . . . . . . . . . 264
Converting an application into an OLE container . . . . . . . . . . . . . . . . . . . . 264
Converting application into a linking and embedding server . . . . . . . . . . . . . . 265
Adding automation support . . . . . . . . . 265
Other useful topics . . . . . . . . . . . . . . . . 265
Learning about ObjectComponents. . . . . . 265
What is OLE? . . . . . . . . . . . . . . . . . . . . 266
Linking and embedding . . . . . . . . . . . . 266
Automation. . . . . . . . . . . . . . . . . . . . . 267
What does OLE look like? . . . . . . . . . . . . 267
Inserting an object . . . . . . . . . . . . . . . . 268
Editing an object in place . . . . . . . . . . . 269
Activating, deactivating, and selecting an object. . . . . . . . . . . . . . . . . . . . . . . 271
Finding an object's verbs . . . . . . . . . . . . 272
Linking an object. . . . . . . . . . . . . . . . . . 273
Opening an object to edit it. . . . . . . . . . . 274
What is ObjectComponents? . . . . . . . . . . 275
OLE 2 features supported by ObjectComponents . . . . . . . . . . . . . . . 276
Using ObjectComponents . . . . . . . . . . . . 278
Overview of classes and messages . . . . . . 278
Linking and embedding classes . . . . . . . 279
Connector objects. . . . . . . . . . . . . . . . . 279
Automation classes. . . . . . . . . . . . . . . . 280
ObjectComponents messages. . . . . . . . . 281
Messages and windows . . . . . . . . . . . . 283
New ObjectWindows OLE classes. . . . . . 283
Exception handling in ObjectComponents. . . 284
TXOle and OLE error codes. . . . . . . . . . 285
Building an ObjectComponents application . . . . . . . . . . . . . . . . . . . . . 286
Distributing files with your application . . . . 287
How ObjectComponents works . . . . . . . . 287
How ObjectComponents talks to OLE . . . . 287
How ObjectComponents talks to you . . . . 288
Linking and embedding connections. . . . . 288
Automation connections . . . . . . . . . . . . 291

ObjectComponents Programming Tools . . . 292
Utility programs . . . . . . . . . . . . . . . . . . 292
Where do I look for information? . . . . . . . 293
Books . . . . . . . . . . . . . . . . . . . . . . . . 293
Online Help . . . . . . . . . . . . . . . . . . . . 294
Example programs . . . . . . . . . . . . . . . 294
Glossary of OLE terms . . . . . . . . . . . . . 295

Chapter 19
**Creating an OLE container 303**

Turning a Doc/View application into an OLE container . . . . . . . . . . . . . . . . . . . 303
1. Connecting objects to OLE . . . . . . . . . . 304
Deriving the application object from TOcModule . . . . . . . . . . . . . . . . . . .304
Inheriting from OLE classes . . . . . . . . . .305
Creating an application dictionary . . . . . .306
2. Registering a container . . . . . . . . . . . . 306
Building registration tables . . . . . . . . . . .307
Understanding registration macros . . . . . .309
Creating a registrar object. . . . . . . . . . . .310
3. Supporting OLE commands . . . . . . . . . 311
Setting up the Edit menu and the tool bar. . . . . . . . . . . . . . . . . . . . . . .312
Loading and saving compound documents. . . . . . . . . . . . . . . . . . . . .312
4. Building the container. . . . . . . . . . . . . 313
Including OLE headers . . . . . . . . . . . . .314
Compiling and linking . . . . . . . . . . . . .314
Turning an ObjectWindows application into an OLE container . . . . . . . . . . . . . 315
1. Setting up the application. . . . . . . . . . . 316
Defining an application dictionary object . .316
Modifying your application class . . . . . . .316
2. Registering a container . . . . . . . . . . . . 317
Creating registration tables . . . . . . . . . . .317
Creating a registrar object. . . . . . . . . . . .318
3. Setting up the client window. . . . . . . . . 319
Inheriting from OLE classes . . . . . . . . . .320
Delaying the creation of the client window in SDI applications . . . . . . . . .320
Creating ObjectComponents view and document objects. . . . . . . . . . . . . . . .321
4. Programming the user interface . . . . . . . 322
Handling OLE-related messages and events . . . . . . . . . . . . . . . . . . . . . .322
Supporting menu merging . . . . . . . . . . .324
Updating the Edit menu . . . . . . . . . . . .327
Assigning a tool bar ID . . . . . . . . . . . . .328
5. Building a container . . . . . . . . . . . . . . 328
Including OLE headers . . . . . . . . . . . . .328
Compiling and linking . . . . . . . . . . . . .328
Turning a C++ application into an OLE container . . . . . . . . . . . . . . . . . . . . . 328
1. Registering a container . . . . . . . . . . . . .329
Building a registration table. . . . . . . . . . .330
Creating the registrar object. . . . . . . . . . .330
Creating a memory allocator . . . . . . . . . .331
2. Creating a view window. . . . . . . . . . . . .332
Creating, resizing, and destroying the view window . . . . . . . . . . . . . . . . . .332
Creating a TOcDocument and TOcView . . .333
Handling WM_OCEVENT . . . . . . . . . . .334
Handling selected view events . . . . . . . . .335
Painting the document. . . . . . . . . . . . . .335
Activating and deactivating objects . . . . . .337
3. Programming the main window . . . . . . .337
Creating the main window . . . . . . . . . . .337
Handling WM_OCEVENT . . . . . . . . . . .337
Handling selected application events . . . . .338
Handling standard OLE menu commands. . . . . . . . . . . . . . . . . . . . .338
Building the program . . . . . . . . . . . . . . .339
Including ObjectComponents headers . . . .339
Compiling and linking. . . . . . . . . . . . . .340

Chapter 20
**Creating an OLE server 341**

Turning a Doc/View application into an OLE server . . . . . . . . . . . . . . . . . . . . 341
1. Connecting objects to OLE. . . . . . . . . . .342
Creating an application dictionary. . . . . . .342
Deriving the application object from TOcModule . . . . . . . . . . . . . . . . . . .343
Inheriting from OLE classes. . . . . . . . . . .344
2. Registering a linking and embedding server . . . . . . . . . . . . . . . . . . . . . . . .344
Building registration tables . . . . . . . . . . .344
Creating a registrar object. . . . . . . . . . . .347
Processing the command line. . . . . . . . . .349
3. Drawing, loading, and saving objects . . . .350
Telling clients when an object changes . . . .350
Loading and saving the server's documents. . . . . . . . . . . . . . . . . . . . .351
4. Building the server . . . . . . . . . . . . . . .351
Including OLE headers . . . . . . . . . . . . .351
Compiling and linking. . . . . . . . . . . . . .352
Turning an ObjectWindows application into an OLE server . . . . . . . . . . . . . . . 352
1. Registering the server . . . . . . . . . . . . .353
Creating an application dictionary. . . . . . .353
Creating registration tables . . . . . . . . . . .353
Creating the document list . . . . . . . . . . .355
Creating the registrar object. . . . . . . . . . .355
2. Setting up the client window . . . . . . . . .356
Creating helper objects for a document. . . .356
3. Modifying the application class. . . . . . . .357

Understanding the TRegLink document list. . . . 358
4. Building the server. . . . 359
Including OLE headers . . . . 359
Compiling and linking . . . . 359
Turning a C++ application into an OLE server . . . . 359
1. Creating a memory allocator . . . . 360
2. Registering the application . . . . 360
Building registration tables . . . . 361
Creating the document list . . . . 361
Creating the registrar object . . . . 362
Writing the factory callback function . . . . 363
3. Creating a view window . . . . 366
Creating, resizing, and destroying the view window . . . . 366
Creating a new server document . . . . 367
Handling WM_OCEVENT . . . . 368
Handling selected view events . . . . 369
Painting the document . . . . 369
4. Programming the main window . . . . 370
Creating the main window . . . . 370
Handling WM_OCEVENT . . . . 370
Handling selected application events . . . . 371
5. Building the server. . . . 371
Including ObjectComponents headers . . . . 371
Compiling and linking . . . . 371
Understanding registration . . . . 372
Storing information in the registration database . . . . 372
Registering localized entries . . . . 373
Registering custom entries . . . . 374
Making a DLL server. . . . 374
Pros and cons of DLL servers . . . . 374
Building a DLL server . . . . 375
Updating your document registration table . . . . 375
Compiling and linking . . . . 377
Debugging a DLL server . . . . 377
Tools for DLL servers . . . . 379
Registering your DLL server . . . . 379
Running your DLL server . . . . 379

## Chapter 21
## Automating an application 381

Steps to automating an application . . . . 381
1. Registering an automation server . . . . 382
Creating a registration table . . . . 382
Creating a registrar object. . . . 385
2. Declaring automatable methods and properties . . . . 385
Writing declaration macros. . . . 386
Providing optional hooks for validation and filtering . . . . 387
3. Defining external methods and properties . . . . 388
Writing definition macros . . . . 389
Data type specifiers in an automation definition. . . . 390
Exposing data for enumeration. . . . 392
4. Building the server . . . . 393
Including header files . . . . 394
Compiling and linking. . . . 394
Enhancing automation server functions. . . . 394
Combining multiple C++ objects into a single OLE automation object . . . . 395
Telling OLE when the object goes away . . . . 396
Localizing symbol names. . . . 397
Putting translations in the resource script . . . . 397
Marking translatable strings in the source code . . . . 398
Understanding how ObjectComponents uses XLAT resources . . . . 399
Localizing registration strings . . . . 400
Exposing collections of objects. . . . 400
Constructing and exposing a collection class. . . . 401
Implementing an iterator for the collection. . . . 402
Adding other members to the collection class . . . . 404
Creating a type library . . . . 405

## Chapter 22
## Creating an automation controller 407

Steps for building an automation controller . . . . 407
Including header files . . . . 408
Creating a TOleAllocator object . . . . 408
Declaring proxy classes . . . . 408
Implementing proxy classes . . . . 410
Specifying arguments in a proxy method. . . . 411
Creating and using proxy objects . . . . 412
Compiling and linking . . . . 412
Enumerating automated collections . . . . 413
Declaring a proxy collection class . . . . 413
Implementing the proxy collection class . . . . 414
Declaring a collection property . . . . 415
Sending commands to the collection . . . . 415

## Appendix A
## Converting ObjectWindows code 417

Converting your code . . . . 418
Converting to Borland C++ 4.5 . . . . 418
OWLCVT conversions . . . . 419
OWLCVT command-line syntax . . . . 420

Backing up your old source files . . . . . . . . 420
How to use OWLCVT from the command line . . . . . . . . . . . . . . . . . . . . . . . . 420
How to use OWLCVT in the IDE . . . . . . . 422
Conversion checklist . . . . . . . . . . . . . . . 422
Conversion procedures . . . . . . . . . . . . . 424
Handling messages and events. . . . . . . . . 424
Removing DDVT functions. . . . . . . . . . .425
Naming conventions . . . . . . . . . . . . 426
Adding an event response table declaration . . . . . . . . . . . . . . . . . . .426
Adding an event response table definition . . . . . . . . . . . . . . . . . . . .426
Adding event response table entries . . . . .427
Responding to command messages. . . . 427
Responding to child ID-based messages . . . . . . . . . . . . . . . . . . 427
Responding to notification messages . . . 428
Responding to general messages . . . . . 428
Event response table samples . . . . . . . . .429
Changing your window objects . . . . . . . . 431
Converting constructors . . . . . . . . . . . .431
Calling Windows API functions . . . . . . . . 432
Changing header files . . . . . . . . . . . . . . 433
Using the new header file locations . . . . . .433
Using the new streamlined ObjectWindows header files . . . . . . . . .433
ObjectWindows resources. . . . . . . . . . . . 434
Compiling resources. . . . . . . . . . . . . . .434
Menu resources. . . . . . . . . . . . . . . . . .435
Constructing virtual bases. . . . . . . . . . . . 435
Downcasting virtual bases to derived types . . . . . . . . . . . . . . . . . . . . . . . . 435
Moving from Object-based containers to the BIDS library . . . . . . . . . . . . . . . . . 436
Streaming . . . . . . . . . . . . . . . . . . . . . 437
Removed insertion and extraction operators. . . . . . . . . . . . . . . . . . . . . .437
Implementing streaming . . . . . . . . . . . .437
MDI classes . . . . . . . . . . . . . . . . . . . . 438
Making the frame and client . . . . . . . . . .439
Making a child window. . . . . . . . . . . . .440
WB_MDICHILD . . . . . . . . . . . . . . . . .440
Relocated functions . . . . . . . . . . . . . . .440
Replacing ActiveChild with GetActiveChild . . . . . . . . . . . . . . . . .441
MainWindow variable . . . . . . . . . . . . . .441
Using a dialog as the main window . . . . . .441
TApplication message processing functions. . . . . . . . . . . . . . . . . . . . . .442
GetModule function . . . . . . . . . . . . . . .443
DefXXXProc functions . . . . . . . . . . . . . .443
Overriding. . . . . . . . . . . . . . . . . . . . .444
Using DefWndProc for registered messages. . . . . . . . . . . . . . . . . . . .444
Paint function. . . . . . . . . . . . . . . . . . . .444
CloseWindow, ShutDownWindow, and Destroy functions . . . . . . . . . . . . . . .445
ForEach and FirstThat functions. . . . . . . . .445
TComboBoxData and TListBoxData classes. .446
TEditWindow and TFileWindow classes . . .446
Using the OLDFILEW example . . . . . . . .447
Adding TEditSearch and TEditFile client windows. . . . . . . . . . . . . . . . . . . .447
TSearchDialog and TFileDialog classes. . . . .448
ActivationResponse function . . . . . . . . . .448
Dispatch-handling functions. . . . . . . . . . .448
DispatchAMessage function . . . . . . . . . . .449
General messages. . . . . . . . . . . . . . . . .449
The DefProc parameter . . . . . . . . . . . . .449
Command messages. . . . . . . . . . . . . . .449
KBHandlerWnd . . . . . . . . . . . . . . . . . .449
MAXPATH . . . . . . . . . . . . . . . . . . . . .450
Style conventions . . . . . . . . . . . . . . . . .450
Changing WinMain to OwlMain. . . . . . . .450
Data types and names . . . . . . . . . . . . . .451
Replacing MakeWindow with Create. . . . .451
Replacing ExecDialog with Execute . . . . . .452
Getting the application and module instance. . . . . . . . . . . . . . . . . . . . . .452
Defining WIN30, WIN31, and STRICT . . . .452
Troubleshooting. . . . . . . . . . . . . . . . . . 452
OWLCVT errors . . . . . . . . . . . . . . . . . .452
Compiler warnings . . . . . . . . . . . . . . . .453
Compiler errors . . . . . . . . . . . . . . . . . .453
Run-time errors . . . . . . . . . . . . . . . . . .454
**Index 457**

# Tables

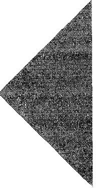

1.1 Data member inheritance . . . . . . . . . . . . . 7
1.2 ObjectWindows-encapsulated device contexts . . . . . . . . . . . . . . . . . . . . . . . 12
1.3 GDI support classes . . . . . . . . . . . . . . . 13
4.1 Command message macros . . . . . . . . . . . 43
4.2 Message macros . . . . . . . . . . . . . . . . . . 44
4.3 Sample message macros and function names . . . . . . . . . . . . . . . . . . . . . . . 45
4.4 Child notification message macros . . . . . . 46
6.1 Specialized exception classes . . . . . . . . . . 63
6.2 ObjectWindows exception-handling macro expansion . . . . . . . . . . . . . . . . . . . . . 64
7.1 Window creation attributes . . . . . . . . . . . 67
7.2 Default window attributes . . . . . . . . . . . 68
7.3 Default window attributes . . . . . . . . . . . 71
7.4 Standard MDI child-window menu behavior . . . . . . . . . . . . . . . . . . . . . . 83
8.1 TMenu constructors for creating menu objects . . . . . . . . . . . . . . . . . . . . . . . . 85
8.2 TMenu constructors for modifying menu objects . . . . . . . . . . . . . . . . . . . . . . . . 86
8.3 TMenu constructors for querying menu objects . . . . . . . . . . . . . . . . . . . . . . . . 87
8.4 TMenuDescr constructors . . . . . . . . . . . . 93
9.1 ObjectWindows-encapsulated dialog boxes . . . . . . . . . . . . . . . . . . . . . . . 106
9.2 Common dialog box TData members . . . . 108
9.3 Common dialog box TData members . . . . 108
9.4 Color common dialog box TData data members . . . . . . . . . . . . . . . . . . . . . 109
9.5 Font common dialog box TData data members . . . . . . . . . . . . . . . . . . . . . 110
9.6 File open and save common dialog box TData data members . . . . . . . . . . . . . . . 111
9.7 Printer common dialog box TData data members . . . . . . . . . . . . . . . . . . . . . 115
10.1 Document manager's File menu . . . . . . . 120
10.2 Document creation mode flags . . . . . . . . 123
10.3 Predefined Doc/View event handlers . . . 138
10.4 Doc/View property attributes . . . . . . . . 140
11.1 Controls and their ObjectWindows classes . . . . . . . . . . . . . . . . . . . . . . . 143
11.2 TListBox member functions for modifying list boxes . . . . . . . . . . . . . . . . . . . . . 148
11.3 TListBox member functions for querying list boxes . . . . . . . . . . . . . . . . . . . . . 149
11.4 List box notification messages . . . . . . . . 149
11.5 TCheckBox member functions for modifying selection boxes . . . . . . . . . . . 153
11.6 TCheckBox member functions for querying selection boxes . . . . . . . . . . . . 154
11.7 Notification codes and TScrollBar member functions . . . . . . . . . . . . . . . . 157
11.8 Pure virtual functions in TSlider . . . . . . . 158
11.9 TEdit member functions and Edit menu commands . . . . . . . . . . . . . . . . . . . . . 160
11.10 TEdit member functions for querying edit controls . . . . . . . . . . . . . . . . . . . . . 161
11.11 TEdit member functions for modifying edit controls . . . . . . . . . . . . . . . . . . . . . 161
11.12 Summary of combo box styles . . . . . . . . 162
11.13 TComboBox member functions for modifying combo boxes . . . . . . . . . . . . 164
11.14 TComboBox member functions for querying combo boxes . . . . . . . . . . . . . 164
11.15 Transfer buffer members for each type of control . . . . . . . . . . . . . . . . . . . . . . . 166
11.16 TListBoxData data members . . . . . . . . . . 166
11.17 TListBoxData member functions . . . . . . . 166
11.18 TComboBoxData data members . . . . . . . 167
11.19 TComboBoxData member functions . . . . . 167
11.20 Transfer flag parameters . . . . . . . . . . . . 169
12.1 Hint mode flags . . . . . . . . . . . . . . . . . 186
17.1 Allowable library combinations . . . . . . . . 261
18.1 How to add container support to an existing application . . . . . . . . . . . . . . . 264
18.2 How to add server support to an existing application . . . . . . . . . . . . . . . . . . . . 265
18.3 Some ObjectComponents classes used for linking and embedding . . . . . . . . . . . . . 279
18.4 Some ObjectComponents classes used for automation . . . . . . . . . . . . . . . . . . . 280
18.5 Application messages for TOcApp clients . . . . . . . . . . . . . . . . . . . . . . . . 281
18.6 View messages for TOcView and TOcRemView clients . . . . . . . . . . . . . 282
18.7 New classes in ObjectWindows for OLE support . . . . . . . . . . . . . . . . . . . . . . . 283
18.8 ObjectComponents exception classes . . . . 284
18.9 Libraries for building ObjectComponents programs . . . . . . . . . . . . . . . . . . . . . . 286
18.10 Descriptions of the ObjectComponents chapters in this book . . . . . . . . . . . . . . 293
18.11 Online Help files with information about ObjectComponents and OLE . . . . . . . . . 294
19.1 Non-OLE classes and the corresponding classes that add OLE support . . . . . . . . . 305
19.2 Keys a container registers to support linking and embedding . . . . . . . . . . . . . 308

19.3 Commands an OLE container places on its Edit menu. . . . . . . . . . . . . . . . . . . . 312
19.4 Libraries for building ObjectComponents programs. . . . . . . . . . . . . . . . . . . . . . 314
19.5 Standard message handlers providing OLE functionality. . . . . . . . . . . . . . . . 322
20.1 Keys a linking and embedding server registers. . . . . . . . . . . . . . . . . . . . . . . 346
20.2 Command-line switches that ObjectComponents recognizes . . . . . . . . 349
21.1 Keys an automation server registers . . . . . 384
21.2 Automation data types . . . . . . . . . . . . . 391
21.3 Enumerable C++ types and the automation types for exposing them. . . . . 393
22.1 Macros for implementing proxy object member functions . . . . . . . . . . . . . . . . 411
22.2 Message response member functions and event response tables . . . . . . . . . . . . . . 427

# Figures


1.1 TDialog inheritance . . . . . 7
2.1 The basic ObjectWindows application . . . . 18
2.2 First-instance and each-instance initialization . . . . . 22
2.3 Dialog box using the Borland Custom Controls Library . . . . . 28
2.4 Dialog box using the Microsoft 3-D Controls Library . . . . . 28
3.1 Interface elements vs. interface objects . . . . 29
4.1 Window message processing . . . . . 42
5.1 Button gadget states . . . . . 56
7.1 Example layout windows . . . . . 70
7.2 Sample decorated frame window . . . . . 79
7.3 Sample MDI application . . . . . 81
8.1 Menu descriptor application without child windows open . . . . . 91
8.2 Menu descriptor application with child windows open . . . . . 91
10.1 Doc/View model diagram . . . . . 118
18.1 The Edit menu in the sample program SdiOle . . . . . 268
18.2 The Insert Object dialog box . . . . . 269
18.3 A newly inserted object being edited in place . . . . . 270
18.4 The same inserted object after being medited . . . . . 270
18.5 The container's restored user interface after the object becomes inactive . . . . . 271
18.6 The speed menu for a selected object . . . . . 272
18.7 The Insert Object dialog box just before inserting a linked object . . . . . 273
18.8 The new verb list for the newly linked object . . . . . 274
18.9 An object opened for editing . . . . . 275
18.10 How applications interact with OLE through ObjectComponents . . . . . 276
18.11 How the ObjectComponents connector objects are related . . . . . 280
18.12 How objects in your application interact with ObjectComponents . . . . . 290
18.13 How TServedObject connects an automated class to OLE . . . . . 292

# Introduction



ObjectWindows 2.5 is the Borland C++ application framework for Windows 3.1, Win32s, and Windows NT. ObjectWindows lets you build full-featured Windows applications quickly and easily. ObjectWindows 2.5 provides the following features:

- Easy creation of OLE 2.0 applications, including containers, servers, and automated applications, using the ObjectComponents Framework
- Doc/View classes for easy data abstraction and display
- Ease of portability between 16- and 32-bit platforms
- Automated message cracking
- Robust exception and error handling
- Allows easy porting to other compilers and environments because it doesn't use proprietary compiler and language extensions
- Encapsulation of Windows GDI objects
- Printer and print preview classes
- Support for Visual Basic controls, including the only available support for using Visual Basic controls in 32-bit environments
- Input validators

## ObjectWindows documentation

The ObjectWindows 2.5 documentation set consists of the *ObjectWindows Programmer's Guide* (this manual), the *ObjectWindows Reference Guide*, and the *ObjectWindows Tutorial*.

The *ObjectWindows Reference Guide* presents a comprehensive, alphabetical listing and description of all ObjectWindows classes, their member functions, data members, and so on. The *ObjectWindows Reference Guide* should be your reference for specific technical data about an ObjectWindows class or function.

The *ObjectWindows Tutorial* contains a tutorial on how to build a basic ObjectWindows application utilizing many of the ObjectWindows library's key features. If you're new to ObjectWindows, or if there are features with which you're not familiar, you should

follow the steps in the *ObjectWindows Tutorial* to learn how to program using ObjectWindows.

## Programmer's Guide organization

The *ObjectWindows Programmer's Guide* presents topics in a task-oriented fashion, describing how to use functional groups of ObjectWindows classes to accomplish various tasks. The manual is organized as follows:

This chapter, **Introduction**, introduces you to ObjectWindows 2.5 and directs you to other chapters of the book for more information.

**Chapter 1, "Overview of ObjectWindows,"** presents a brief, nontechnical overview of the ObjectWindows hierarchy.

**Chapter 2, "Application and module objects,"** describes application objects and the application class *TApplication*.

**Chapter 3, "Interface objects,"** discusses the use of interface objects in the ObjectWindows 2.5 programming model. Interface objects are instances of classes representing windows, dialog boxes, and controls; these classes are based on the class *TWindow*.

**Chapter 4, "Event handling,"** explains response tables, the ObjectWindows 2.5 method for event handling.

**Chapter 5, "Command enabling,"** describes the ObjectWindows 2.5 command-enabling mechanism for enabling and disabling command items such as menu choices and control bar buttons, setting menu item text, and checking and unchecking command items.

**Chapter 6, "ObjectWindows exception handling,"** describes the ObjectWindows 2.5 exception-handling mechanism.

**Chapter 7, "Window objects,"** describes window objects, including how to use frame windows, layout windows, decorated frame windows, and MDI windows.

**Chapter 8, "Menu objects,"** discusses the use of menu objects and the *TMenu* class.

**Chapter 9, "Dialog box objects,"** explains how to use dialog box objects (such as *TDialog* and *TDialog*-derived objects) and also Windows common dialog boxes, which are based on the *TCommonDialog* class.

**Chapter 10, "Doc/View objects,"** presents the ObjectWindows 2.5 Doc/View programming model, which uses the *TDocument*, *TView*, and *TDocManager* classes.

**Chapter 11, "Control objects,"** discusses the use of various controls, such as buttons, list boxes, edit boxes, and so on.

**Chapter 12, "Gadget and gadget window objects,"** explains gadgets and gadget windows, including control bars, status bars, button gadgets, and so on.

**Chapter 13, "Printer objects,"** describes how to use the printer and print preview classes.

**Chapter 14, "Graphics objects,"** presents the classes that encapsulate Windows GDI.

**Chapter 15, "Validator objects,"** describes the use of input validators in edit controls.

**Chapter 16, "Visual Basic controls,"** discusses using Visual Basic controls and the *TVbxControl* class in your ObjectWindows application.

**Chapter 17, "ObjectWindows dynamic-link libraries,"** explains the use of ObjectWindows-encapsulated dynamic-link libraries (DLLs).

**Chapter 18, "Support for OLE in Borland C++,"** presents an overview of the ObjectComponents encapsulation of OLE capabilities.

**Chapter 19, "Creating an OLE container,"** describes how to make a container application whose compound documents can hold linked and embedded OLE objects.

**Chapter 20, "Creating an OLE server,"** describes how to make a server application that creates data objects for containers to link or embed.

**Chapter 21, "Automating an application,"** describes what a program must do in order to let other programs control it through automation.

**Chapter 22, "Creating an automation controller,"** describes the steps a program must take in order to manipulate automation objects.

**Appendix A, "Converting ObjectWindows code,"** describes how to convert your ObjectWindows 1.0 applications so they work properly in ObjectWindows 2.5.

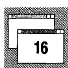
# Typefaces and icons used in this book

The following table shows the special typographic conventions used in this book.

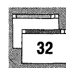
| Typeface | Meaning |
|---|---|
| **Boldface** | Boldface type indicates language keywords (such as **char**, **switch**, and **begin**) and command-line options (such as **–rn**). |
| *Italics* | Italic type indicates program variables and constants that appear in text. This typeface is also used to emphasize certain words, such as new terms. |
| `Monospace` | Monospace type represents text as it appears onscreen or in a program. It is also used for anything you must type literally (such as `TD32` to start up the 32-bit Turbo Debugger). |
| Menu \| Command | This command sequence represents a choice from the menu bar followed by a menu choice. For example, the command "File \| Open" represents the Open command on the File menu. |

**Note** This icon indicates material you should take special notice of.

This manual also uses the following icons to indicate sections that pertain to specific operating environments:



16-bit Windows



32-bit Windows

Chapter 1

# Overview of ObjectWindows

This chapter presents an overview of the ObjectWindows 2.5 hierarchy. It also describes the basic groupings of the ObjectWindows 2.5 classes, explains how each class fits together with the others, and refers you to specific chapters for more detailed information about how to use each class.

## Working with class hierarchies

This section describes some of the basic properties of classes, focusing specifically on ObjectWindows classes. It covers the following topics:

- What you can do with a class
- Inheriting members
- Types of member functions

### Using a class

There are three basic things you can do with a class:

- Derive a new class from it
- Add its behavior to that of another class
- Create an instance of it (instantiate it)

#### Deriving new classes

To change or add behavior to a class, you derive a new class from it:

```
class TNewWindow : public TWindow
{
  public:
    TNewWindow(...);
  // ...
};
```

When you derive a new class, you can do three things:

- Add new data members
- Add new member functions
- Override inherited member functions

Adding new members lets you add to or change the functionality of the base class. You can define a new constructor for your derived class to call the base classes' constructors and initialize any new data members you might have added.

## Mixing object behavior

With ObjectWindows designed using multiple inheritance, you can derive new classes that inherit the behavior of more than one class. Such "mixed" behavior is different from the behavior you get from single inheritance derivation. Instead of inheriting the behavior of the base class and being able to add to and change it, you're inheriting *and combining* the behavior of several classes.

As with single inheritance derivation, you can add new members and override inherited ones to change the behavior of your new class.

## Instantiating classes

To use a class, you must create an instance of it. There are a number of ways you can instantiate a class:

- You can use the standard declaration syntax. This is the same syntax you use to declare any standard variable such as an **int** or **char**. In this example, *app* is initialized by calling the *TMyApplication* constructor with no arguments:

  ```
  TMyApplication app;
  ```

  You can use this syntax only when the class has a default constructor or a constructor in which all the parameters have default values.

- You can also use the standard declaration syntax along with arguments to call a particular constructor. In this example, *app* is initialized by calling the *TMyApplication* constructor with a **char** * argument:

  ```
  TMyApplication app("AppName");
  ```

- You can use the **new** operator to allocate space for and instantiate an object. For example:

  ```
  TMyApplication *app;
  app = new TMyApplication;
  ```

- You can also use the **new** operator along with arguments. In this example, *app* is initialized by calling the *TMyApplication* constructor with a **char** * argument:

  ```
  TMyApplication* app = new TMyApplication("AppName");
  ```

The constructors call the base class' constructors and initialize any needed data members. You can only instantiate classes that aren't abstract; that is, classes that don't contain a pure virtual function.

## Abstract classes

Abstract classes, which are classes with pure virtual member functions that you must override to provide some behavior, serve two main purposes. They provide a conceptual framework to build other classes on and, on a practical level, they reduce coding effort.

For example, the ObjectWindows *THSlider* and *TVSlider* classes could each be derived directly from *TScrollBar*. Although one is vertical and the other horizontal, they have similar functionality and responses. This commonality warrants creating an abstract class called *TSlider*. *THSlider* and *TVSlider* are then derived from *TSlider* with the addition of a few specialized member functions to draw the sliders differently.

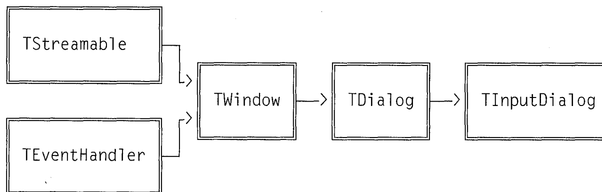
You can't create an instance of an abstract class. Its pure virtual member functions must be overridden to make a useful instance. *TSlider*, for example, doesn't know how to paint itself or respond directly to mouse events.

If you wanted to create your own slider (for example, a circular slider), you might try deriving your slider from *TSlider* or it might be easier to derive from *THSlider* or *TVSlider*, depending on which best meets your needs. In any case, you add data members and add or override member functions to add the desired functionality. If you wanted to have diagonal sliders going both northwest-southeast and southwest-northeast, you might want to create an intermediate abstract class called *TAngledSlider*.

# Inheriting members

The following figure shows the inheritance of *TInputDialog*. As you can see, *TInputDialog* is derived from *TDialog*, which is derived from *TWindow*, which is in turn derived from *TEventHandler* and *TStreamable*. Inheritance lets you add more specialized behavior as you move further along the hierarchy.

**Figure 1.1** TDialog inheritance




The following table shows the public data members of each class, including those inherited from the *TDialog* and *TWindow* base classes:

**Table 1.1** Data member inheritance

| TWindow | TDialog | TInputDialog |
|---|---|---|
| Status | *Status* | *Status* |
| HWindow | *HWindow* | *HWindow* |
| Title | *Title* | *Title* |
| Parent | *Parent* | *Parent* |
| Attr | *Attr* | *Attr* |

**Table 1.1** Data member inheritance (continued)

| TWindow | TDialog | TInputDialog |
|---|---|---|
| DefaultProc | *DefaultProc* | *DefaultProc* |
| Scroller | *Scroller* | *Scroller* |
| | *IsModal* | *IsModal* |
| | | *Prompt* |
| | | *Buffer* |
| | | *BufferSize* |

*TInputDialog* inherits all the data members of *TDialog* and *TWindow* and adds the data members it needs to be an input dialog box.

To fully understand what you can do with *TInputDialog*, you have to understand its inheritance: a *TInputDialog* object is both a dialog box (*TDialog*) and a window (*TWindow*). *TDialog* adds the concept of modality to the *TWindow* class. *TInputDialog* extends that by adding the ability to store and retrieve user-input data.

# Types of member functions

There are four (possibly overlapping) types of ObjectWindows member functions:

- Virtual
- Nonvirtual
- Pure virtual
- Default placeholder

## Virtual functions

Virtual functions can be overridden in derived classes. They differ from pure virtual functions in that they don't *have* to be overridden in order to use the class. Virtual functions provide you with *polymorphism*, which is the ability to provide a consistent class interface, even when the functionality of your classes is quite different.

## Nonvirtual functions

You should not override nonvirtual functions. Therefore, it's important to make virtual any member function that derived classes might need to override (an exception is the event-handling functions defined in your response tables). For example, *TWindow::CanClose* is virtual because derived classes should override it to verify whether the window should close. On the other hand, *TWindow::SetCaption* is nonvirtual because you usually don't need to change the way a window's caption is set.

The problem with overriding nonvirtual functions is that classes that are derived from your derived class might try to use the overridden function. Unless the new derived classes are *explicitly* aware that you have changed the functionality of the derived function, this can lead to faulty return values and run-time errors.

### Pure virtual functions

You must override pure virtual functions in derived classes. Functions are marked as pure virtual using the `= 0` initializer. For example, here's the declaration of *TSlider::PaintRuler*:

```
virtual void PaintRuler(TDC& dc) = 0;
```

You must override all of an abstract class' pure virtual functions in a derived class before you can create an instance of that derived class. In most cases, when using the standard ObjectWindows classes, you won't find this to be much of a problem; most of the ObjectWindows classes you might need to derive from are *not* abstract classes. In lieu of pure virtual functions, many ObjectWindows classes use default placeholder functions.

### Default placeholder functions

Unlike pure virtual functions, default placeholder functions don't have to be overridden. They offer minimal default actions or no actions at all. They serve as placeholders, where you can place code in your derived classes. For example, here's the definition of *TWindow::EvLButtonDblClk*:

```
inline void
TWindow::EvLButtonDblClk (uint modKeys, TPoint &)
{
  DefaultProcessing();
}
```

By default, *EvLButtonDblClk* calls *DefaultProcessing* to perform the default message processing for that message. In your own window class, you could override *EvLButtonDblClk* by defining it in your class' response table. Your version of *EvLButtonDblClk* can provide some custom behavior you want to happen when the user clicks the left mouse button. You can also continue to provide the base class' default processing by calling the base class' version of the function.

## Object typology

The ObjectWindows hierarchy has many different types of classes that you can use, modify, or add to. You can separate what each class does into the following groups:

- Windows
- Dialog boxes
- Controls
- Graphics
- Printing
- Modules and applications
- Doc/View applications
- Miscellaneous Windows elements

## Window classes

An important part of any Windows application is, of course, the window. ObjectWindows provides several different window classes for different types of windows (not to be confused with the Windows "window class" registration types):

- Windows
- Frame windows
- MDI windows
- Decorated windows

Chapter 7 describes the window classes in detail.

### Windows

*TWindow* is the base class for all window classes. It represents the functionality common to all windows, whether they are dialog boxes, controls, MDI windows, or so on.

### Frame windows

*TFrameWindow* is derived from *TWindow* and adds the functionality of a frame window that can hold other client windows.

### MDI windows

Multiple Document Interface (MDI) is the Windows standard for managing multiple documents or windows in a single application. *TMDIFrame*, *TMDIClient*, and *TMDIChild* provide support for MDI in ObjectWindows applications.

### Decorated windows

Several classes, such as *TLayoutWindow* and *TLayoutMetrics*, work together to provide support for *decoration* controls like tool bars, status bars, and message bars. Using multiple inheritance, decoration support is added into frame windows and MDI frame windows in *TDecoratedFrame* and *TDecoratedMDIFrame*.

## Dialog box classes

*TDialog* is a derived class of *TWindow*. It's used to create dialog boxes that handle a variety of user interactions. Dialog boxes typically contain controls to get user input. Dialog box classes are explained in detail in Chapter 9.

### Common dialog boxes

In addition to specialized dialog boxes your own application might use, ObjectWindows supports Windows' common dialog boxes for:

- Choosing files (*TFileOpenDialog*, and *TFileSaveDialog*)
- Choosing fonts (*TChooseFontDialog*)
- Choosing colors (*TChooseColorDialog*)
- Choosing printing options (*TPrintDialog*)
- Searching and replacing text (*TFindDialog*, and *TReplaceDialog*)

### Other dialog boxes

ObjectWindows also provides additional dialog boxes that aren't based on the Windows common dialog boxes:

- Inputting text (*TInputDialog*)
- Aborting print jobs (*TPrinterAbortDlg*, used in conjunction with the *TPrinter* and *TPrintout* classes)

## Control classes

*TControl* is a class derived from *TWindow* to support behavior common to all controls. ObjectWindows offers four types of controls:

- Standard Windows controls
- Widgets
- Gadgets
- Decorations

All these controls are discussed in depth in Chapter 11, except for gadgets, which are discussed in Chapter 12.

### Standard Windows controls

Standard Windows controls include list boxes, scroll bars, buttons, check boxes, radio buttons, group boxes, edit controls, static controls, and combo boxes. Member functions let you manipulate these controls.

### Widgets

Unlike standard Windows controls, ObjectWindows widgets are specialized controls written entirely in C++. The widgets ObjectWindows offers include horizontal and vertical sliders (*THSlider* and *TVSlider*) and gauges (*TGauge*).

### Gadgets

Gadgets are similar to standard Windows controls, in that they are used to gather input from or convey information to the user. But gadgets are implemented differently from controls. Unlike most other interface elements, gadgets are not windows: gadgets don't have window handles, they don't receive events and messages, and they aren't based on *TWindow*.

Instead, gadgets must be contained in a gadget window. The gadget window controls the presentation of the gadget, all message processing, and so on. The gadget receives its commands and direction from the gadget window.

### Decorations

Decorations are specialized child windows that let the user choose a command, provide a place to give the user information, or somehow allow for specialized communication with the user.

- A control bar (*TControlBar*) lets you arrange a set of buttons on a bar attached to a window as shortcuts to using menus (the SpeedBar in the Borland C++ IDE is an example of this functionality).
- A tool box (*TToolBox*) lets you arrange a set of buttons on a floating palette.
- Message bars (*TMessageBar*) are bars, usually at the bottom of a window, where you can display information to the user. For example, the Borland C++ IDE uses a message bar to give you brief descriptions of what menu commands and SpeedBar buttons do as you press them.
- Status bars (*TStatusBar*) are similar to message bars, but have room for more than one piece of information. The status bar in the Borland C++ IDE shows your position in the edit window, whether you're in insert or overtype mode, and error messages.

## Graphics classes

Windows offers a powerful but complex graphics library called the Graphics Device Interface (GDI). ObjectWindows encapsulates GDI to make it easier to use device context (DC) classes (*TDC*) and GDI objects (*TGDIObject*).

See Chapter 14 for full details on these classes.

### DC classes

With GDI, instead of drawing directly on a device (like the screen or a printer), you draw on a bitmap using a device context (DC). A *device context* is a collection of tools, settings, and device information regarding a graphics device and its current drawing state. This allows for a high degree of device independence when using GDI functions. The following table lists the different types of DCs that ObjectWindows encapsulates.

**Table 1.2** ObjectWindows-encapsulated device contexts

| Type of device context | ObjectWindows DC class |
|---|---|
| Memory | *TMemoryDC* |
| Metafile | *TMetaFileDC* |
| Bitmap | *TDibDC* |
| Printer | *TPrintDC* |
| Window | *TWindowDC* |
| Desktop | *TDesktopDC* |
| Screen | *TScreenDC* |
| Client | *TClientDC* |
| Paint | *TPaintDC* |

## GDI objects

*TGDIObject* is a base class for several other classes that represent things you can use to draw with and to control drawings. The following table lists these classes and other ObjectWindows GDI support classes.

**Table 1.3** GDI support classes

| Type of GDI object | ObjectWindows GDI class |
|---|---|
| Pens | *TPen* |
| Brushes | *TBrush* |
| Fonts | *TFont* |
| Palettes | *TPalette* |
| Bitmaps | *TBitmap, TDib, TUIBitmap* |
| Icons | *TIcon* |
| Cursors | *TCursor* |
| Regions | *TRegion* |
| Points | *TPoint* |
| Size | *TSize* |
| Rectangles | *TRect* |
| Color specifiers | *TColor* |
| RGB triple color | *TRgbTriple* |
| RGB quad color | *TRgbQuad* |
| Palette entries | *TPaletteEntry* |
| Metafile | *TMetafilePict* |

# Printing classes

*TPrinter* makes printing significantly easier by encapsulating the communications with printer drivers. *TPrintout* encapsulates the task of printing a document. Chapter 13 discusses how to use the printing classes.

# Module and application classes

A Windows application is responsible for initializing windows and ensuring that messages Windows sends to it are sent to the proper window. ObjectWindows encapsulates that behavior in *TApplication*. A DLL's behavior is encapsulated in *TModule*. For full details on module and application objects, see Chapter 2.

# Doc/View classes

The Doc/View classes are a complete abstraction of a generic document-view model. The base classes of the Doc/View model are *TDocManager*, *TDocument*, and *TView*. The Doc/View model is a system in which data is contained in and accessed through a document object, and displayed and manipulated through a view object. Any number of views can be associated with a particular document type. You can use this to display the same data in a number of different ways.

For example, you can display a line both graphically (as a line in a window) and as sets of numbers indicating the coordinates of the points that make up the line. This would require one document that contains the data and two view classes: one view class to display the line onscreen and another view class to display the coordinates of the points in the line. You can also modify the data through the views so that, in this case, you could change the data in the line by either drawing in the graphical display or by typing in numbers to modify and add coordinates in the numerical display.

The Doc/View model is discussed in depth in Chapter 10.

## Miscellaneous classes

Since Windows is so varied, not all the classes ObjectWindows provides fall into neat categories. This section discusses those miscellaneous classes.

### Menus

Menus can be static or you can modify them or even load whole new menus. *TMenu* and its derived classes (*TSystemMenu* and *TPopupMenu*) let you easily manipulate menus. Chapter 8 discusses the menu classes in more detail.

### Clipboard

The Windows Clipboard is one of the main ways users share data between applications. ObjectWindows' *TClipboard* object lets you easily provide Clipboard support in your applications. See Chapter 7 for details.

Chapter

2

# Application and module objects

This chapter describes how to use application objects, including

- Deriving an application object from the *TApplication* class
- Creating an application object
- Overriding base class functions in derived application objects to customize application behavior
- Using the Borland Custom Control and Microsoft Control 3-D libraries, including automatically subclassing custom controls as Microsoft Control 3-D controls

ObjectWindows encapsulates Windows applications and DLL modules using the *TApplication* and *TModule* classes, respectively. *TModule* objects

- Encapsulate the initialization and closing functions of a Windows DLL
- Contain the *hInstance* and *lpCmdLine* parameters, which are equivalent to the parameters of the same name that are passed to the *WinMain* function in a non-ObjectWindows application (note that both *WinMain* and *LibMain* have these two parameters in common)

*TApplication* objects build on the basic functionality provided by *TModule*. *TApplication* and *TApplication*-derived objects

- Encapsulate the initialization, run-time management, and closing functions of a Windows application
- Contain the values of the *hPrevInstance* and *nCmdShow* parameters, which are equivalent to the parameters of the same name that are passed to the *WinMain* function in a non-ObjectWindows application

The *TApplication* class is derived from the *TModule* class. You usually won't need to create a *TModule* object yourself, unless you're working with a DLL. See Chapter 17 for more information on using DLLs in an ObjectWindows application.

# The minimum requirements

To use a *TApplication* object, you must first:

- Include the correct header file
- Create an application object
- Call the application object's *Run* function

## Including the header file

*TApplication* is defined in the header file owl\applicat.h; you must include this header file to use *TApplication*. Because *TApplication* is derived from *TModule*, owl\applicat.h includes owl\module.h.

## Creating an object

You can create a *TApplication* object using one of two constructors. The most commonly used constructor is this:

```
TApplication(const char far* name);
```

This version of the *TApplication* constructor takes a string, which becomes the application's name. If you don't specify a name, by default the constructor names it the null string. *TApplication* uses this string as the application name.

The second version of the *TApplication* constructor lets you specify a number of parameters corresponding to the parameters normally passed to the *WinMain* function:

```
TApplication(const char far* name,
             HINSTANCE instance,
             HINSTANCE prevInstance,
             const char far* cmdLine,
             int cmdShow);
```

You can use this constructor to pass command parameters to the *TApplication* object. This is discussed on page 20.

## Calling the Run function

The most obvious thing that *TApplication::Run* function does is to start your application running. But in doing so it performs a number of other very important tasks, including

- Initializing the application
- Creating and displaying the main window
- Running the application's message loop

Each of these tasks is discussed later in this chapter. For the purposes of creating the basic ObjectWindows application, however, it is sufficient to know that *Run* is the function you call to make your application go.

## Finding the object

You may need to access an application object from outside that object's scope. For example, you may need to call one of the application object's member functions from a function in a derived window class. But because the window object is not in the same scope as the application object, you have no way of accessing the application object. In this case, you must find the application object.

*TApplication* contains several member functions and data members you might need to call from outside the scope your application object. To find these easily, the *TWindow* class has a member function, *GetApplication,* that returns a pointer to the application object. You can then use this pointer to call *TApplication* member functions and access *TApplication* data members. The following listing shows a possible use of *GetApplication.*

```
void
TMyWindow::Error()
{
  // display message box containing the application name
  MessageBox("An error occurred!",
             GetApplication()->Name, MB_OK);
}
```

The *TWindow* class is discussed in Chapter 7.

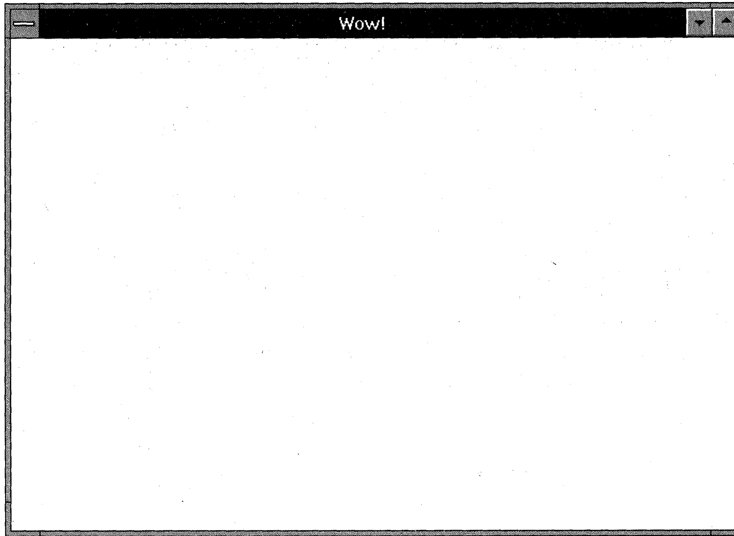
## Creating the minimum application

Here's the smallest ObjectWindows application you can create. It includes the correct header file, creates a *TApplication* object, and calls that object's *Run* function.

```
#include <owl\applicat.h>

int
OwlMain(int argc, char* argv[])
{
  return TApplication("Wow!").Run();
}
```

This creates a Windows application with a main window with the caption "Wow!" You can resize, move, minimize, maximize, and close this window. In a real application, you'd derive a new class for the application to add more functionality. Notice that the only function you have to call explicitly in this example is the *Run* function. Figure 2.1 shows how this application looks when it's running.

Figure 2.1 The basic ObjectWindows application




# Initializing applications

Initializing an ObjectWindows application takes four steps:

- Constructing the application object
- Initializing the application
- Initializing each new instance
- Initializing the main window

## Constructing the application object

When you construct a *TApplication* object, it calls its *InitApplication*, *InitInstance*, and *InitMainWindow* member functions to start the application. You can override any of those members to customize how your application initializes. Since the base *InitMainWindow* function only creates a default window object with no way to customize its functionality, you must override *InitMainWindow* to start creating an application with the functionalty you want to create. To override a function in *TApplication* you need to derive your own application class from *TApplication*.

The constructor for the *TApplication*-derived class *TMyApplication* shown in the following examples takes the application name as its only argument; its default value is zero, for no name. The application name is used for the default main window title and in error messages. The application name is referenced by a **char far** * member of the *TModule* base class called *Name*. You can set the application name one of two ways:

- Your application class' constructor can explicitly call *TApplication*'s constructor, passing the application name onto *TApplication*. The following example shows this method:

```
#include <owl\applicat.h>

class TMyApplication: public TApplication
{
  public:
    // This constructor initializes the base class constructor
    TMyApplication(const char far* name = 0) : TApplication(name) {}
    ⋮
};
```

- Override one of *TApplication*'s initialization functions, usually *InitMainWindow*, and set the application name there. The following example shows this method:

```
#include <owl\applicat.h>

class TMyApplication: public TApplication
{
  public:
    // This constructor just uses the default base class constructor
    TMyApplication(const char far* name = 0) {}
    void InitMainWindow()
    {
      if (name)
      {
        Name = new char[strlen(name) + 1];
        strcpy(Name, name);
      }
    }
};
```

ObjectWindows applications don't require an explicit *WinMain* function; the ObjectWindows libraries provide one that performs error handling and exception handling. You can perform any initialization you want in the *OwlMain* function, which is called by the default *WinMain* function.

To construct an application object, create an instance of your application class in the *OwlMain* function. The following example shows a simple application object's definition and instantiation:

```
#include <owl\applicat.h>

class TMyApplication: public TApplication
{
public:
  TMyApplication(const char far* name = 0): TApplication(name) {}
};

int
OwlMain(int argc, char* argv[])
{
  return TMyApplication("Wow!").Run();
}
```

## Using WinMain and OwlMain

ObjectWindows furnishes a default *WinMain* function that provides extensive error checking and exception handling. This *WinMain* function sets up the application and calls the *OwlMain* function.

Although you can use your own *WinMain* by placing it in a source file, there's little reason to do so. Everything you would otherwise do in *WinMain* you can do in *OwlMain* or in *TApplication* initialization member functions. The following example shows a possible use of *OwlMain* in an application. *OwlMain* checks to see whether the user specified any parameters on the application's command line. If so, *OwlMain* creates the application object using the first parameter as the application name. If not, *OwlMain* creates the application object using Wow! as the application name.

```
#include <owl\applicat.h>
#include <string.h>

class TMyApplication: public TApplication
{
public:
  TMyApplication(const char far* name = 0) : TApplication(name) {}
};

int
OwlMain(int argc, char* argv[])
{
  char title[30];
  if(argc >= 2)
    strcpy(title, argv[1]);
  else
    strcpy(title, "Wow!");
  return TMyApplication(title).Run();
}
```

If you do decide to provide your own *WinMain, TApplication* supports passing traditional *WinMain* function parameters with another constructor. The following example shows how to use that constructor to pass *WinMain* parameters to the *TApplication* object:

```
#include <owl\applicat.h>

class TMyApplication : public TApplication
{
public:
  TMyApplication (const char far* name,
                  HINSTANCE instance,
                  HINSTANCE prevInstance,
                  const char far* cmdLine,
                  int cmdShow)
    : TApplication (name, instance, prevInstance, cmdLine, cmdShow) {}
};

int
```

```
PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpszCmdLine, int nCmdShow)
{
  return TMyApplication("MyApp", hInstance, hPrevInstance,
                        lpszCmdLine, nCmdShow).Run();
}
```

## Calling initialization functions

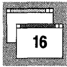
*TApplication* contains three initialization functions:

- *InitApplication* initializes the first instance of the application
- *InitInstance* initializes each instance of the application
- *InitMainWindow* initializes the application's main window

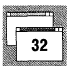
How these functions are called depends on whether this is the first instance of the application. *InitApplication* is called only for the first instance of the application on the system. *InitInstance* is the next function called for the first instance. It is the first function called by additional instances. *InitInstance* calls *InitMainWindow.*

If the application is a 32-bit application, each instance appears to be the first instance of the application, affecting this chain of execution. This is described in the next section.

### Initializing the application




Users can run multiple copies of an application simultaneously. From the point of view of a 16-bit application, first-instance initialization happens only when another copy of the application is not currently running. Each-instance initialization happens every time the user runs the application. If a user starts and closes your application, starts it again, and so on, each instance is a first instance because the instances don't run at the same time.




In the case of 32-bit applications, each application runs in its own address space, with no shared instance data, so that each instance appears as a first instance. Therefore every time you start a 32-bit application, it performs both first-instance initialization and each-instance initialization.

If the current instance is a first instance (indicated by the data member *hPrevInstance* being set to zero), *InitApplication* is called. You can override *InitApplication* in your derived application class; the default *InitApplication* has no functionality.

For example, you could use first-instance initialization to make the main window's caption indicate whether it's the first instance. To do this,

1. Add a data member called *WindowTitle* in your derived application class.
2. In the application class' constructor, set *WindowTitle* to "Additional Instance."
3. Override *InitApplication* to set *WindowTitle* to "First Instance."

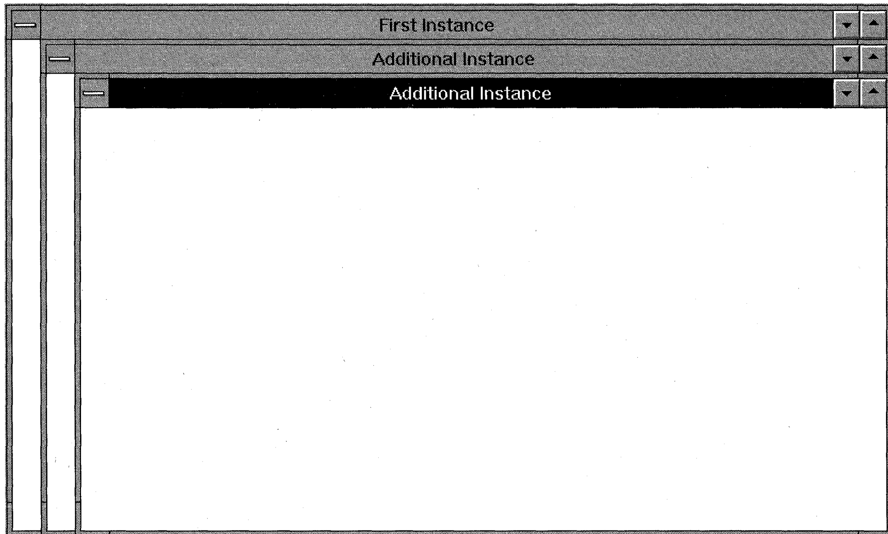
If your application is the first instance of the application, *InitApplication* is called and overwrites the value of *WindowTitle* that was set in the constructor. The following example shows how the code might look:

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

```
#include <cstring.h>

class TTestApp : public TApplication
{
  public:
    TTestApp() : TApplication("Instance Tester"), WindowTitle("Additional Instance") {}

  protected:
    string WindowTitle;

    void InitApplication() { WindowTitle = string("First Instance"); }
    void InitMainWindow() { SetMainWindow(new TFrameWindow(0, WindowTitle.c_str())); }
};

int
OwlMain(int /* argc */, char* /* argv */[])
{
  return TTestApp().Run();
}
```

Figure 2.2 shows a number of instances of this application open on the desktop. Note that the first instance—the upper left one—has the title "First Instance," while every other instance has the title "Additional Instance."

**Figure 2.2** First-instance and each-instance initialization




Again, this application doesn't function as you might expect when it's built as a 32-bit application. Because each instance of a 32-bit application perceives itself to be the first instance of the application, multiple copies running at the same time would all have the caption "First Instance."

## Initializing each new instance

A user can run multiple instances (copies) of an application simultaneously. You can override *TApplication::InitInstance* to perform any initialization you need to do for each instance.

*InitInstance* calls *InitMainWindow* and then creates and shows the main window you set up in *InitMainWindow*. If you override *InitInstance*, be sure your new *InitInstance* calls *TApplication::InitInstance*. The following example shows how to use *InitInstance* to load an accelerator table:

```
void
TTestApp::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(MAKEINTRESOURCE(MYACCELS));
}
```

## Initializing the main window

By default, *TApplication::InitMainWindow* creates a frame window with the same name as the application object. This window isn't very useful, because it can't receive or process any user input. You must override *InitMainWindow* to create a window object that does process user input.

Normally, your *InitMainWindow* function creates a *TFrameWindow* or *TFrameWindow*-derived object and calls the *SetMainWindow* function. *SetMainWindow* takes one parameter, a pointer to a *TFrameWindow* object, and returns a pointer to the old main window (if this is a new application that hasn't yet set up a main window, the return value is zero). Chapter 7 describes window classes and objects in detail.

The following example shows a simple application that creates a *TFrameWindow* object and makes it the main window:

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

class TMyApplication: public TApplication
{
public:
  TMyApplication(): TApplication() {}
  void InitMainWindow();
};

void
TMyApplication::InitMainWindow()
{
  // Just sets the main window with a basic TFrameWindow object
  SetMainWindow(new TFrameWindow(0, "My First Main Window"));
}

int
OwlMain(int argc, char* argv[])
{
```

```
  return TMyApplication("Wow!").Run();
}
```

When you run this application, the caption bar is titled "My First Main Window," and not "Wow!". The application name passed in the *TApplication* constructor is used only when you do not provide a main window. Once again, this example doesn't do a lot; there is still no provision for the frame window to process any user input. But once you have derived a window class that does interact with the user, you use the same simple method to display the window.

## Specifying the main window display mode

You can change how your application's main window is displayed by setting the *TApplication* data member *nCmdShow*, which corresponds to the *WinMain* parameter *nCmdShow*. You can set this variable as soon as the *Run* function begins, up until the time you call *TApplication::InitInstance*. This effectively means you can set *nCmdShow* in either the *InitApplication* or *InitMainWindow* function.

For example, suppose you want to display your window maximized whenever the user runs the application. You could set *nCmdShow* in your *InitMainWindow* function:

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

class TMyApplication : public TApplication
{
  public :
    TMyApplication(char far *name) : TApplication(name) {}
    void InitMainWindow();
};

void
TMyApplication::InitMainWindow()
{
  // Sets the main window
  SetMainWindow(new TFrameWindow(0, "Maximum Window"));

  // Sets nCmdShow so that the window is maximized when it's created
  nCmdShow = SW_SHOWMAXIMIZED;
}

int
OwlMain(int argc, char* argv[])
{
  return TMyApplication("Wow!").Run();
}
```

*nCmdShow* can be set to any value appropriate as a parameter to the *ShowWindow* Windows function or the *TWindow::Show* member function, such as SW_HIDE, SW_SHOWNORMAL, SW_NORMAL, and so on.

### Changing the main window

You can use the *SetMainWindow* function to change your main window during the course of your application. *SetMainWindow* takes one parameter, a pointer to a *TFrameWindow* object, and returns a pointer to the old main window (if this is a new application that hasn't yet set up a main window, the return value is zero). You can use this pointer to keep the old main window in case you want to restore it. Alternatively, you can use this pointer to delete the old main window object.

## Application message handling

Once your application is initialized, the application object's *MessageLoop* starts running. *MessageLoop* is responsible for processing incoming messages from Windows. There are two ways you can refine message processing in an ObjectWindows application:

- Extra message processing, by overriding default message handling functions
- Idle processing

### Extra message processing

*TApplication* has member functions that provide the message-handling functionality for any ObjectWindows application. These functions are *MessageLoop*, *IdleAction*, *PreProcessMenu*, and *ProcessAppMsg*. See the *ObjectWindows Reference Guide* for more information.

### Idle processing

Idle processing lets your application take advantage of the idle time when there are no messages waiting (including user input). If there are no waiting messages, *MessageLoop* calls *IdleAction*.

To perform idle processing, override *IdleAction* to perform the actual idle processing. Remember that idle processing takes place while the user isn't doing anything. Therefore, idle processing should be short-lasting. If you need to do anything that takes longer than a few tenths of a second, you should split it up into several processes.

*IdleAction*'s parameter (*idleCount*) is a **long** specifying the number of times *IdleAction* was called between messages. You can use *idleCount* to choose between low-priority and high-priority idle processing. If *idleCount* reaches a high value, you know that a long period without user input has passed, so it's safe to perform low-priority idle processing.

Return **true** from *IdleAction* to call *IdleAction* back sooner.

You should always call the base class *IdleAction* function in addition to performing your own processing. If you're writing applications for Windows NT, you can also use multiple threads for background processing.

# Closing applications

Users usually close a Windows application by choosing File | Exit or pressing *Alt+F4*. It's important, though, that the application be able to intercept such an attempt, to give the user a chance to save any open files. *TApplication* lets you do that.

## Changing closing behavior

*TApplication* and all window classes have or inherit a member function *CanClose*. Whenever an application tries to shut down, it queries the main window's and document manager's *CanClose* function. (The exception to this is when dialog boxes are cancelled by the user clicking the Cancel button or pressing *Esc*; in which case, the dialog box is simply destroyed, bypassing the *CanClose* function.) If either of the application object or the document manager has children, it calls the *CanClose* function for each child. In turn, each child calls the *CanClose* function of each of their children if any, and so on.

The *CanClose* function gives each object a chance to prepare to be shut down. It also gives the object a chance to cancel the shutdown if necessary. When the object has completed its clean-up procedure, its *CanClose* function should return **true**.

If any of the *CanClose* functions called returns **false**, the shut-down procedure is cancelled.

### Closing the application

The *CanClose* mechanism gives the application object, the main window, and any other windows a chance to either prepare for closing or prevent the closing from taking place. In the end, the application object approves the closing of the application. The normal closing sequence looks like this:

1. Windows sends a WM_CLOSE message to the main window.
2. The main window object's *EvClose* member function calls the application object's *CanClose* member function.
3. The application object's *CanClose* member function calls the main window object's *CanClose* member function.
4. The main window and document manager objects call *CanClose* for each of their child windows. The main window and document manager objects' *CanClose* functions return **true** only if all child windows' *CanClose* member functions return **true**.
5. If both the main window and document manager objects' *CanClose* functions return **true**, the application object's *CanClose* function returns **true**.
6. If the application object's *CanClose* function returns **true**, the *EvClose* function shuts down the main window and ends the application.

### Modifying CanClose

*CanClose* should rarely return **false**. Instead, *CanClose* gives you a chance to perform any actions necessary to return **true**. If you override *CanClose* in your derived application

objects, the function should return **false** *only* if it's unable to do something necessary for orderly shutdown or if the user wants to keep the application running.

For example, suppose you are creating a text editor. A possible procedure to follow in the *CanClose* member function would be to:

1. Check to see if the editor text had changed.
2. If so, prompt the user to ask whether the text should be saved before closing, using a message box with Yes, No, and Cancel buttons.
3. Check the return value from the message box:
   - If the user clicks Yes, save the file, then return **true** from the *CanClose* function.
   - If the user clicks No, simply return **true** from the *CanClose* function without saving the file.
   - If the user clicks Cancel, indicating the user doesn't want to close the application yet, return **false** from the *CanClose* function without saving the file.

# Using control libraries

*TApplication* has functions for loading the Borland Custom Controls Library (BWCC.DLL for 16-bit applications and BWCC32.DLL for 32-bit applications) and the Microsoft 3-D Controls Library (contained in the file CTL3DV2.DLL for 16-bit applications and CTL3D32.DLL for 32-bit applications). These DLLs are widely used to provide a standard look-and-feel for many applications.

## Using the Borland Custom Controls Library

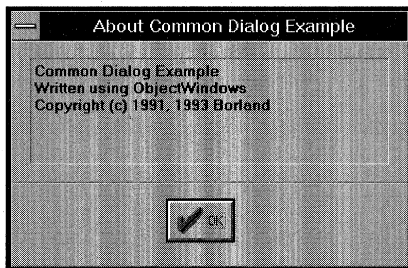
You can open and close the Borland Custom Controls Library using the function *TApplication::EnableBWCC*. *EnableBWCC* takes one parameter, a **bool**, and returns a **void**. When you pass **true** to *EnableBWCC*, the function loads the DLL if it's not already loaded. When you pass **false** to *EnableBWCC*, the function unloads the DLL if it's not already unloaded.

You can find out if the Borland Custom Controls Library DLL is loaded by calling the function *TApplication::BWCCEnabled*. *BWCCEnabled* takes no parameters. If the DLL is loaded, *BWCCEnabled* returns **true**; if not, *BWCCEnabled* returns **false**.

Once the DLL is loaded, you can use all the regular functionality of Borland Custom Controls Library. *EnableBWCC* automatically opens the correct library regardless of whether you have a 16- or a 32-bit application.

Figure 2.3 shows an example of a dialog box using the Borland Custom Controls Library.

**Figure 2.3** Dialog box using the Borland Custom Controls Library

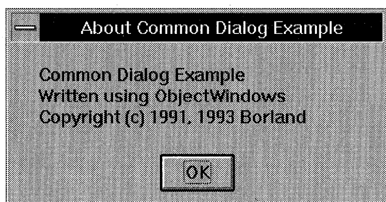



## Using the Microsoft 3-D Controls Library

You can load and unload the Microsoft 3-D Controls Library using the function *TApplication::EnableCtl3d*. *EnableCtl3d* takes one parameter, a **bool**, and returns a **void**. When you pass **true** to *EnableCtl3d*, the function loads the DLL if it's not already loaded. When you pass **false** to *EnableCtl3d*, the function unloads the DLL if it's not already unloaded.

Figure 2.4 shows an example of a dialog box using the Microsoft 3-D Controls Library.

**Figure 2.4** Dialog box using the Microsoft 3-D Controls Library




You can find out if the Microsoft 3-D Controls Library DLL is loaded by calling the function *TApplication::Ctl3dEnabled*. *Ctl3dEnabled* takes no parameters. If the DLL is loaded, *Ctl3dEnabled* returns **true**; if not, *Ctl3dEnabled* returns **false**.

To use the *EnableCtl3dAutosubclass* function, load the Microsoft 3-D Controls Library DLL using *EnableCtl3d*. *EnableCtl3dAutosubclass* takes one parameter, a **bool**, and returns a **void**. When you pass **true** to *EnableCtl3dAutosubclass*, autosubclassing is turned on. When you pass **false** to *EnableCtl3dAutosubclass*, autosubclassing is turned off.

When autosubclassing is on, any non-ObjectWindows dialog boxes you create have a 3-D effect. You can turn autosubclassing off immediately after creating the dialog box; it is not necessary to leave it on when displaying the dialog box.

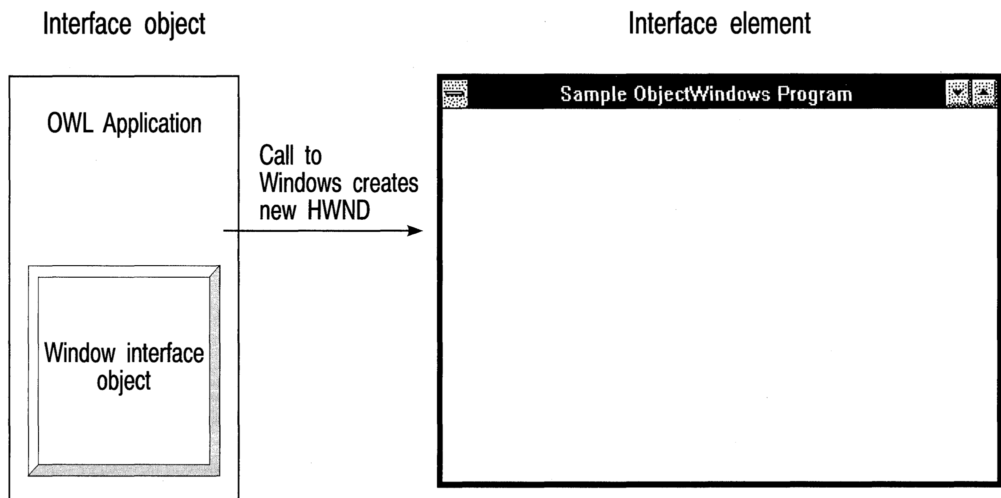
# Chapter 3

# Interface objects

Instances of C++ classes representing windows, dialog boxes, and controls are called *interface objects*. This chapter discusses the general requirements and behavior of interface objects and their relationship with the *interface elements*—the actual windows, dialog boxes, and controls that appear onscreen.

The following figure illustrates the difference between interface objects and interface elements:

**Figure 3.1** Interface elements vs. interface objects




Notice how the interface object is actually inside the application object. The interface object is an ObjectWindows class that is created and stored on the application's heap or stack, depending on how the object is allocated. The interface element, on the other hand, is actually a part of Windows. It is the actual window displayed on the screen.

The information in this chapter applies to all interface objects. This chapter also explains the relationships between the different interface objects of an application, and describes the mechanism that interface objects use to respond to Windows messages.

# Why interface objects?

One of the greatest difficulties of Windows programming is that controlling interface elements can be inconsistent and confusing. Sometimes you send a message to a window; other times you call a Windows API function. The conventions for similar types of operations often differ when those operations are performed with different kinds of elements.

ObjectWindows alleviates much of this difficulty by providing objects that encapsulate the interface elements. This insulates you from having to deal directly with Windows and provides a more uniform interface for controlling interface elements.

## What do interface objects do?

An interface object provides member functions for creating, initializing, managing, and destroying its associated interface element. The member functions manage many of the details of Windows programming for you.

Interface objects also encapsulate the data needed to communicate with the interface element, such as handles and pointers to child and parent windows.

The relationship between an interface object and an interface element is similar to that between a file on disk and a C++ stream object. The stream object only represents an actual file on disk; you manipulate that file by manipulating the stream object. With ObjectWindows, interface objects represent the interface elements that Windows itself actually manages. You work with the object, and Windows takes care of maintaining the Windows element.

# The generic interface object: TWindow

ObjectWindows' interface objects are all derived from *TWindow*, which defines behavior common to all window, dialog box, and control objects. Classes like *TFrameWindow*, *TDialog*, and *TControl* are derived from *TWindow* and refine *TWindow*'s generic behavior as needed.

As the common base class for all interface objects, *TWindow* provides uniform ways to:

- Maintain the relationship between interface objects and interface elements, including creating and destroying the objects and elements
- Handle parent-child relationships between interface objects
- Register new Windows window classes

# Creating interface objects

Setting up an interface object with its associated interface element requires two steps:

1. Calling one of the interface object constructors, which constructs the interface object and sets its attributes.
2. Creating the interface element by telling Windows to create the interface object with a new interface element:
   - When creating most interface elements, you call the interface object's *Create* member function. *Create* also indirectly calls *SetupWindow*, which initializes the interface object by creating an interface element, such as child windows.
   - When creating a modal dialog box, you create the interface element by calling the interface object's *Execute* member function. See page 98 for more information on modal dialog boxes.

The association between the interface object and the interface element is maintained by the interface object's *HWindow* data member, a handle to a window.

## When is a window handle valid?

Normally under Windows, a newly created interface element receives a WM_CREATE message from Windows, and responds to it by initializing itself. ObjectWindows interface objects intercept the WM_CREATE message and call *SetupWindow* instead. *SetupWindow* is where you want to perform your own initialization.

**Note** If part of the interface object's initialization requires the interface element's window handle, you must perform that initialization *after* you call the base class' *SetupWindow*. Prior to the time you call the base class' *SetupWindow*, the window and its child windows haven't been created; *HWindow* isn't valid and shouldn't be used. You can easily test the validity of *HWindow*: if it hasn't been initialized, it is set to NULL.

Although it might seem odd that you can't perform all initialization in the interface object's constructor, there's a good reason: once an interface element is created, you can't change many of its characteristics. Therefore, a two-stage initialization is required: before and after the interface element is created.

The interface object's constructor is the place for initialization before the element is created and *SetupWindow* is the place for initialization after the element is created. You can think of *SetupWindow* as the second part of the constructor.

## Making interface elements visible

Creating an object and its corresponding element doesn't mean that you'll see something on the screen. When Windows creates the interface element, Windows checks to see if the element's style includes WS_VISIBLE. If it does, Windows displays the interface element; if it doesn't, the element is created but not displayed onscreen.

*TWindow*'s constructor sets WS_VISIBLE, so most interface objects are visible by default. But if your object loads a resource, that resource's style depends on what is defined in its

resource file. If WS_VISIBLE is turned on in the resource's style, WS_VISIBLE is turned on for the object. If WS_VISIBLE is *not* turned on in the resource's style, WS_VISIBLE is turned *off* in the object's style. You can set WS_VISIBLE and other window styles in the interface object in the *Attr.Style* data member.

For example, if you use *TDialog* to load a dialog resource that doesn't have WS_VISIBLE turned on, you must explicitly turn WS_VISIBLE before attempting to display the dialog using *Create*.

You can find out whether an interface object is visible by calling *IsWindowVisible*. *IsWindowVisible* returns **true** if the object is visible.

At any point after the interface element has been created, you can show or hide it by calling its *Show* member function with a value of **true** or **false**, respectively.

## Object properties

In addition to the attributes of its interface element, the interface object possesses certain attributes as an ObjectWindows object. You can query and change these properties and characteristics using the following functions:

- *SetFlag* sets the specified flag for the object.
- *ClearFlag* clears the specified flag for the object.
- *IsFlagSet* returns **true** if the specified flag is set, **false** if the specified flag is not set.

You can use the following flags with these functions:

- *wfAlias* indicates whether the object is an alias; see page 77.
- *wfAutoCreate* indicates whether automatic creation is enabled for this object.
- *wfFromResource* indicates whether the interface element is loaded from a resource.
- *wfShrinkToClient* indicates whether the frame window should shrink to fit the size of the client window.
- *wfMainWindow* indicates whether the window is the main window.
- *wfPredefinedClass* indicates whether the window is a predefined Windows class.
- *wfTransfer* indicates whether the window can use the data transfer mechanism. See Chapter 11 for transfer mechanism information.

## Window properties

*TWindow* also provides a couple of functions that let you change resources and properties of the interface element. Because *TWindow* provides generic functionality for a large variety of objects, it doesn't provide very specific functions for resource and property manipulation. High-level objects provide much more specific functionality. But that specific functionality builds on and is in addition to the functionality provided by *TWindow*:

- *SetCaption* sets the window caption to the string that you pass as a parameter.

- *GetWindowTextTitle* returns a string containing the current window caption.
- *SetCursor* sets the cursor of the instance, identified by the *TModule* parameter, to the cursor passed as a resource in the second parameter.
- You can set the accelerator table for a window by assigning the resource ID (which can be a string or an integer) to *Attr.AccelTable*. For example, suppose you have an accelerator table resource called MY_ACCELS. You would assign the resource to *Attr.AccelTable* like this:

```
TMyWnd::TMyWnd(const char* title)
{
  Init(0, title);
  Attr.AccelTable = MY_ACCELS; // AccelTable can be assigned
}
```

For more specific information on these functions, refer to the *ObjectWindows Reference Guide*.

# Destroying interface objects

Destroying interface objects is a two-step process:

- Destroying the interface element
- Deleting the interface object

You can destroy the interface element without deleting the interface object, if you need to create and display the interface element again.

## Destroying the interface element

Destroying the interface element is the responsibility of the interface object's *Destroy* member function. *Destroy* destroys the interface elements by calling the *DestroyWindow* API function. When the interface element is destroyed, the interface object's *HWindow* data member is set to zero. Therefore, you can tell if an interface object is still associated with a valid interface element by checking its *HWindow*.

When a user closes a window on the screen, the following things happen:

- Windows notifies the window.
- The window goes through the *CanClose* mechanism to verify that the window should be closed.
- If *CanClose* approves the closing of the window, the interface element is destroyed and the interface object is deleted.

## Deleting the interface object

If you destroy an interface element yourself so that you can redisplay the interface object later, you must make sure that you delete the interface object when you're done with it.

Because an interface object is nothing more than a regular C++ object, you can delete it using the **delete** statement if you've dynamically allocated the object with **new**.

The following code illustrates how to destroy the interface element and the interface object.

```
TWindow *window = new TWindow(0, "My Window");

// ...

window->Destroy();
delete window;
```

# Parent and child interface elements

In a Windows application, interface elements work together through parent-child links. A parent window controls its child windows, and Windows keeps track of the links. ObjectWindows maintains a parallel set of links between corresponding interface objects.

A child window is an interface element that is managed by another interface element. For example, list boxes are managed by the window or dialog box in which they appear. They are displayed only when their parent windows are displayed. In turn, dialog boxes are child windows managed by the windows that create them.

When you move or close the parent window, the child windows automatically close or move with it. The ultimate parent of all child windows in an application is the main window (there are a couple of exceptions: you can have windows and dialog boxes without parents and all main windows are children of the Windows desktop).

## Child-window lists

When you construct a child-window object, you specify its parent as a parameter to its constructor. A child-window object keeps track of its parent through the *Parent* data member. A parent keeps track of its child-window objects in a private data member called *ChildList*. Each parent maintains its list of child windows automatically.

You can access an object's child windows using the window iterator member functions *FirstThat* and *ForEach*. See page 37 for more information on these functions.

## Constructing child windows

As with all interface objects, child-window objects get created in two steps: constructing the interface object and creating the interface element. If you construct child-window objects in the constructor of the parent window, their interface elements are automatically created when the parent is, assuming that automatic creation is enabled for the child windows. By default, automatic creation is enabled for all ObjectWindows objects based on *TWindow*, with the exception of *TDialog*. See page 37 for more information on automatic creation.

For example, the constructor for a window object derived from *TWindow* that contains three button child windows would look like this:

```
TTestWindow::TTestWindow(TWindow *parent, const char far *title)
{
  Init(parent, title);

  button1 = new TButton(this, ID_BUTTON1, "Show",
                        190, 270, 65, 20, false);
  button2 = new TButton(this, ID_BUTTON2, "Hide",
                        275, 270, 65, 20, false);
  button3 = new TButton(this, ID_BUTTON3, "Transfer",
                        360, 270, 65, 20, false);
}
```

Note the use of the **this** pointer to link the child windows with their parent. Interface object constructors automatically add themselves to their parents' child window lists. When an instance of *TTestWindow* is created, the three buttons are automatically displayed in the window.

## Creating child interface elements

If you don't construct child-window objects in their parent window object's constructor, they won't be automatically created and displayed when the parent is. You can then create them yourself using *Create* or, in the case of modal dialog boxes, *Execute*. In this context, creating means instantiating an interface element.

For example, suppose you have two buttons displayed when the main window is created, one labeled Show and the other labeled Hide. When the user presses the Show button, you want to display a third button labeled Transfer. When the user presses the Hide button, you want to remove the Transfer button:

```
class TTestWindow : public TFrameWindow
{
  public:
    TestWindow(TWindow *parent, const char far *title);

    void
    EvButton1()
    {
      if(!button3->HWindow)
        button3->Create();
    }

    void
    EvButton2()
    {
      if(button3->HWindow)
        button3->Destroy();
    }

    void
```

```
    EvButton3()
    {
      MessageBeep(-1);
    }

  protected:
    TButton *button1, *button2, *button3;

  DECLARE_RESPONSE_TABLE(TTestWindow);
};

DEFINE_RESPONSE_TABLE1(TTestWindow, TFrameWindow)
  EV_COMMAND(ID_BUTTON1, EvButton1),
  EV_COMMAND(ID_BUTTON2, EvButton2),
  EV_COMMAND(ID_BUTTON3, EvButton3),
END_RESPONSE_TABLE;

TTestWindow::TTestWindow(TWindow *parent, const char far *title)
{
  Init(parent, title);
  button1 = new TButton(this, ID_BUTTON1, "Show",
                        10, 10, 75, 25, false);
  button2 = new TButton(this, ID_BUTTON2, "Hide",
                        95, 10, 75, 25, false);
  button3 = new TButton(this, ID_BUTTON3, "Transfer",
                        180, 10, 75, 25, false);
  button3->DisableAutoCreate();
}
```

The call to *DisableAutoCreate* in the constructor prevents the Transfer button from being displayed when *TTestWindow* is created. The conditional tests in the *EvButton1* and *EvButton2* functions work by testing the validity of the *HWindow* data member of the *button3* interface object; if the Transfer button is already being displayed, *EvButton1* doesn't try to display it again, and *EvButton2* doesn't try to destroy the Transfer button if it isn't being displayed.

## Destroying windows

Destroying a parent window also destroys all of its child windows. You do not need to explicitly destroy child windows or delete child window interface objects. The same is **true** for the *CanClose* mechanism; *CanClose* for a parent window calls *CanClose* for all its children. The parent's *CanClose* returns **true** only if all its children return **true** for *CanClose*.

When you destroy an object's interface element, it enables automatic creation for all of its children, *regardless* of whether automatic creation was on or off before. This way, when you create the parent, all the children are restored in the state they were in before their parent was destroyed. You can use this to destroy an interface element, and then re-create it in the same state it was in when you destroyed it.

To prevent this, you must explicitly turn off automatic creation for any child objects you don't want to have created automatically.

## Automatic creation

When automatic creation is enabled for a child interface object before its parent is created, the child is automatically created at the same time the parent is created. This is **true** for all the parent object's children.

To explicitly exclude a child window from the automatic create-and-show mechanism, call the *DisableAutoCreate* member function in the child object's constructor. To explicitly add a child window (such as a dialog box, which would normally be excluded) to the automatic create-and-show mechanism, call the *EnableAutoCreate* member function in the child object's constructor.

By default automatic creation is enabled for all ObjectWindows classes except for dialog boxes.

## Manipulating child windows

*TWindow* provides two iterator functions, *ForEach* and *FirstThat*, that let you perform operations on either all the children in the parent's child list or a single child at a time. *TWindow* also provides a number of other functions that let you determine the number of children in the child list, move through them one at a time, or move to the top or bottom of the list.

### Operating on all children: ForEach

You might want to perform some operation on each of a parent window's child windows. The iterator function *ForEach* takes a pointer to a function. The function can be either a member function or a stand-alone function. The function should take a *TWindow* * and a **void** * argument. *ForEach* calls the function once for each child. The child is passed as the *TWindow* *. The **void** * defaults to 0. You can use the **void** * to pass any arguments you want to your function.

After *ForEach* has called your function, you often need to be careful when dealing with the child object. Although the object is passed as a *TWindow* *, it is actually usually a descendant of *TWindow*. To make sure the child object is handled correctly, you should use the DYNAMIC_CAST macro to cast the *TWindow* * to a *TClass* *, where *TClass* is whatever type the child object is.

For example, suppose you want to check all the check box child windows in a parent window:

```
void
CheckTheBox(TWindow* win, void*)
{
  TCheckbox *cb = DYNAMIC_CAST(win, TCheckbox);
  if(cb)
    cb->Check();
}

void
TMDIFileWindow::CheckAllBoxes()
{
```

```
    ForEach(CheckTheBox);
}
```

If the class you're downcasting to (in this case from a *TWindow* to a *TCheckbox*) is virtually derived from its base, you *must* use the DYNAMIC_CAST macro to make the assignment. In this case, *TCheckbox* isn't virtually derived from *TWindow*, making the DYNAMIC_CAST macro superfluous in this case.

DYNAMIC_CAST returns 0 if the cast could not be performed. This is useful here, because not all of the children are necessarily of type *TCheckbox*. If a child of type *TControlBar* was encountered, the value of *cb* would be 0, thus assuring that you don't try to check a control bar.

### Finding a specific child

You might also want to perform a function only on a specific child window. For example, if you wanted to find the first check box that's checked in a parent window with several check boxes, you would use *TWindow::FirstThat*:

```
bool
IsThisBoxChecked(TWindow* cb, void*)
{
  return cb ? (cb->GetCheck == BF_CHECKED) : false;
}

TCheckBox*
TMDIFileWindow::GetFirstChecked()
{
  return FirstThat(IsThisBoxChecked);
}
```

### Working with the child list

In addition to the iterator functions *ForEach* and *FirstThat*, *TWindow* provides a number of functions that let you locate and manipulate a single child window:

- *NumChildren* returns an **unsigned**. This value indicates the total number of child windows in the child list.
- *GetFirstChild* returns a *TWindow* * that points to the first entry in the child list.
- *GetLastChild* returns a *TWindow* * that points to the last entry in the child list.
- *Next* returns a *TWindow* * that points to the next entry in the child list.
- *Previous* returns a *TWindow* * that points to the prior entry in the child list.

## Registering window classes

Whenever you create an interface element from an interface object using the *Create* or *Execute* functions, the object checks to see if another object of the same type has registered with Windows. If so, the element is created based on the existing Windows registration class. If not, the object automatically registers itself, then is created based on the class just registered. This removes the burden from the programmer of making sure all window classes are registered before use.

Chapter

# 4

# Event handling

This chapter describes how to use ObjectWindows response tables. Response tables are the method you use to handle all events in an ObjectWindows application. There are four main steps to using ObjectWindows response tables:

1. Declare the response table
2. Define the response table
3. Define the response table entries
4. Declare and define the response member functions

To use any of the macros described in this chapter, the header file owl\eventhan.h must be included. This file is already included by owl\module.h (which is included by owl\applicat.h) and owl\window.h, so there is usually no need to explicitly include this file.

ObjectWindows response tables are a major improvement over other methods of handling Windows events and messages, including **switch** statements (such as those in standard C Windows programs) and schemes used in other types of application frameworks. Unlike other methods of event handling, ObjectWindows response tables provide:

- Automatic message "cracking" for predefined command messages, eliminating the need for manually extracting the data encoded in the WPARAM and LPARAM values.
- Compile-time error and type checking, which checks the event-handling function's return type and parameter types.
- Ability to have one function handle multiple messages.
- Support for multiple inheritance, enabling each derived class to build on top of the base class or classes' response tables.
- Portability across platforms by not relying on product-specific compiler extensions.
- Easy handling of command, registered, child ID notification, and custom messages, using the predefined response table macros.

## Declaring response tables

Because the response table is a member of an ObjectWindows class, you must declare the response table when you define the class. ObjectWindows provides the DECLARE_RESPONSE_TABLE macro to hide the actual template syntax that response tables use.

The DECLARE_RESPONSE_TABLE macro takes a single argument, the name of the class for which the response table is being declared. Add the macro at the end of your class definition. For example, *TMyFrame*, derived from *TFrameWindow*, would be defined like this:

```
class TMyFrame : public TFrameWindow
{
  :
  DECLARE_RESPONSE_TABLE(TMyFrame);
};
```

It doesn't matter what the access level is at the point where you declare the response table. That is, it doesn't matter if the declaration is in a position where it would public, protected, or private. The DECLARE_RESPONSE_TABLE macro sets up its own access levels when it's expanded by the preprocessor. By the same token, you must make certain that the DECLARE_RESPONSE_TABLE macro is the last element in your class declaration; otherwise, any members declared after the macro will have unpredictable access levels.

## Defining response tables

Once you've declared a response table, you must define it. Response table definitions must appear outside the class definition.

ObjectWindows provides the DEFINE_RESPONSE_TABLE*X* macro to help define response tables. The value of *X* depends on your class' inheritance, and is a number equal to the number of immediate base classes your class has. END_RESPONSE_TABLE ends the event response table definition.

To define your response table,

1. Begin the response table definition for your class using the DEFINE_RESPONSE_TABLE*X* macro. DEFINE_RESPONSE_TABLE*X* takes *X* + 1 arguments:
   - The name of the class you're defining the response table for
   - The name of each immediate base class
2. Fill in the response table entries (see the next section for information on how to do this step).
3. End the response table definition using the END_RESPONSE_TABLE macro.

For example, the response table definition for *TMyFrame*, derived from *TFrameWindow*, would look like this:

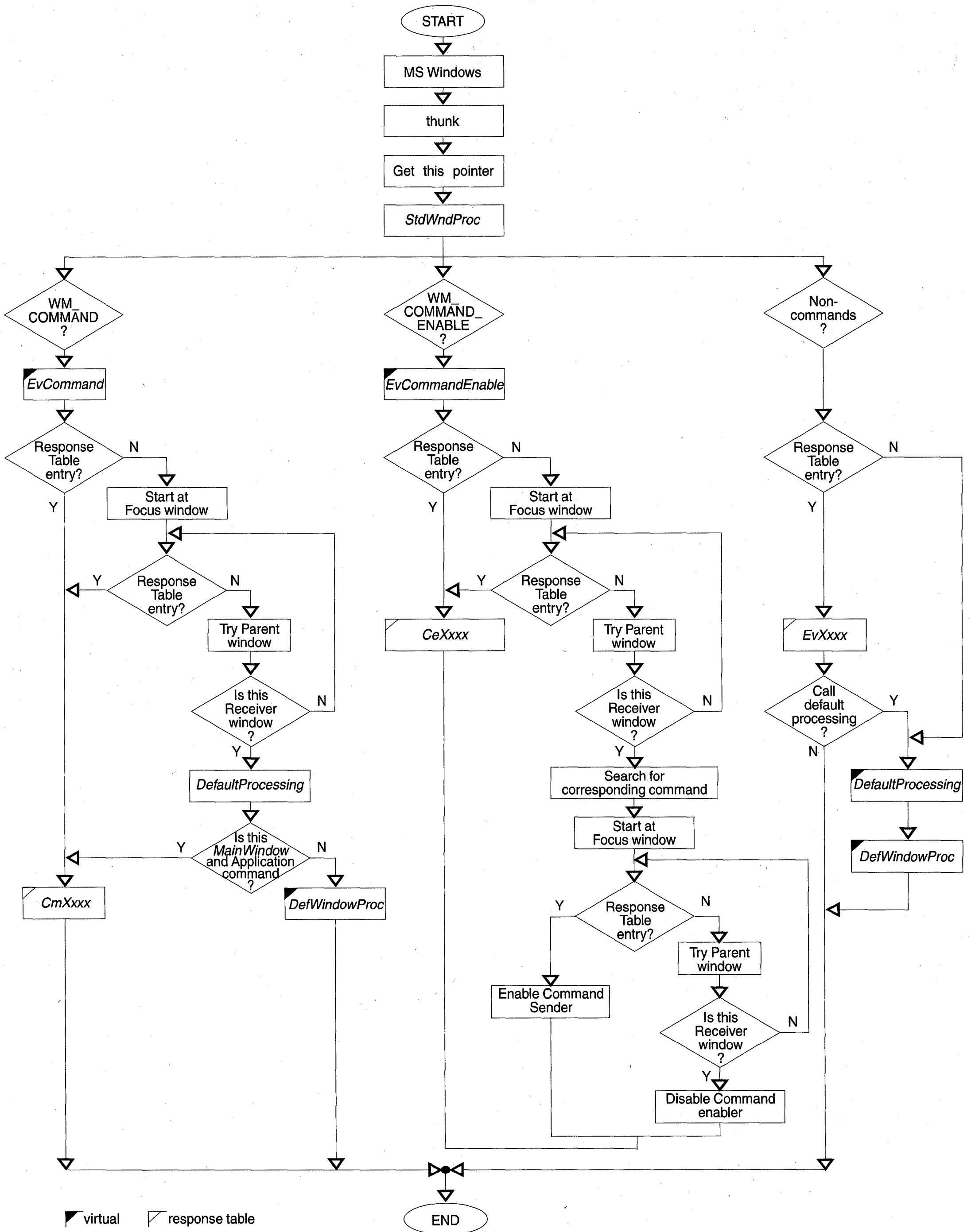
```
DEFINE_RESPONSE_TABLE1(TMyFrame, TFrameWindow)
  EV_WM_LBUTTONDOWN,
  EV_WM_LBUTTONUP,
  EV_WM_MOUSEMOVE,
  EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
```

You must always place a comma after each response table entry and a semicolon after the END_RESPONSE_TABLE macro.

# Defining response table entries

Response table entries associate a Windows event with a particular function. When a window or control receives a message, it checks its response table to see if there is an entry for that message. If there is, it passes the message on to that function. If not, it passes the message up to its parent. If the parent is not the main window, it passes the message up to *its* parent. Once the parent is the main window, it passes the message on to the application object. If the application object doesn't have a response entry for that particular message, the message is handled by ObjectWindows default processing. This is illustrated in Figure 4.1.

**Figure 4.1** Window message processing



ObjectWindows provides a large number of macros for response table entries. These include:

- Command message macros that let you handle command messages and route them to a specified function.
- Standard Windows message macros for handling Windows messages.
- Registered messages (messages returned by *RegisterWindowMessage*).
- Child ID notification macros that let you handle child ID notification codes at the child or the parent.
- Control notification macros that handle messages from specialized controls such as buttons, combo boxes, edit controls, list boxes, and so on.
- Document manager message macros to notify the application that a document or view has been created or destroyed and to notify views about events from the document manager.
- VBX control notifications.

## Command message macros

ObjectWindows provides a large number of macros, called *command message macros*, that let you assign command messages to any function. The only requirement is that the signature of the function you specify to handle a message must match the signature required by the macro for that message. The different types of command message macros and the corresponding function signatures are listed in Table 4.1:

**Table 4.1** Command message macros

| Macro | Prototype | Description |
|---|---|---|
| EV_COMMAND(CMD, UserName) | **void** *UserName*() | Calls *UserName* when the CMD message is received. |
| EV_COMMAND_AND_ID(CMD, UserName) | **void** *UserName*(WPARAM) | Calls *UserName* when the CMD message is received. Passes the command's ID (WPARAM parameter) to *UserName*. |
| EV_COMMAND_ENABLE(CMD, UserName) | **void** *UserName*(TCommandEnabler&) | Used to automatically enable and disable command controls such as menu items, tool bar buttons, and so on. |

There are other message macros that let you pass the raw, unprocessed message on to the event-handling function. These message macros handle any kind of generic message and registered message.

**Table 4.2** Message macros

| Macro | Prototype | Description |
|---|---|---|
| EV_MESSAGE(MSG, UserName) | LRESULT *UserName*(WPARAM, LPARAM) | Calls *UserName* when the user-defined message MSG is received. MSG is passed to *UserName* without modification. |
| EV_REGISTERED(MSG, UserName) | LRESULT *UserName*(WPARAM, LPARAM) | Calls *UserName* when the registered message MSG is received. MSG is passed to *UserName* without modification. |

It is very important that you correctly match the function signature with the macro that you use in the response table definition. For example, suppose you have the following code:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow* parent, const char* name) : TFrameWindow(parent, name) {}

  protected:
    void CmAdvise();

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame, TFrameWindow)
  EV_COMMAND_AND_ID(CM_ADVISE, CmAdvise),
END_RESPONSE_TABLE;

void
TMyFrame::CmAdvise()
{
  ⋮
}
```

This code produces a compile-time error because the EV_COMMAND_AND_ID macro requires a function that returns **void** and takes a single WPARAM parameter. In this example, the function correctly returns **void**, but incorrectly takes no parameters. To make this code compile correctly, change the member declaration and function definition of *TMyFrame::CmAdvise* to:

```
void TMyFrame::CmAdvise(WPARAM cmd);
```

## Windows message macros

ObjectWindows provides predefined macros for all standard Windows messages. You can use these macros to handle standard Windows messages in one of your class' member functions.

To find the name of the macro to handle a particular predefined message, preface the message name with EV_. This macro passes the message on to a function with a predefined name. To determine the function name, remove the WM_ from the message name, add *Ev* to the remaining part of the message name, and convert the name to lowercase with capital letters at word boundaries. Table 4.3 shows some examples.

**Table 4.3** Sample message macros and function names

| Message | Response table macro | Function name |
|---|---|---|
| WM_PAINT | EV_WM_PAINT | *EvPaint* |
| WM_LBUTTONDOWN | EV_WM_LBUTTONDOWN | *EvLButtonDown* |
| WM_MOVE | EV_WM_MOVE | *EvMove* |

The advantage to using these message macros is that the message is automatically "cracked," that is, the parameters that are normally encoded in the LPARAM and WPARAM parameters are broken out into their constituent parts and passed to the event-handling function as individual parameters.

For example, the EV_WM_CTLCOLOR macro passes the cracked parameters to an event-handling function with the following signature:

```
HBRUSH EvCtlColor(HDC hDCChild, HWND hWndChild, uint nCtrlType);
```

Message cracking provides for strict C++ compile-time type checking, and helps you catch errors as you compile your code rather than at run time. It also helps when migrating applications from 16-bit to 32-bit and vice versa. Chapter 3 in the *ObjectWindows Reference Guide* lists each predefined message, its corresponding response table macro, and the signature of the corresponding event-handling function.

To use a predefined Windows message macro:

1 Add the macro to your response table.

2 Add the appropriate member function with the correct name and signature to your class.

3 Define the member function to handle the message however you want.

For example, suppose you wanted to perform some operation when your *TMyFrame* window object received the WM_ERASEBKGND message. The code would look like this:

```
class TMyFrame : public TFrameWindow {
public:
  bool EvEraseBkgnd(HDC);

  DECLARE_RESPONSE_TABLE(TMyFrame);
};
```

```
DEFINE_RESPONSE_TABLE(TMyFrame, TFrameWindow)
  EV_WM_ERASEBKGND,
END_RESPONSE_TABLE;

bool
TMyFrame::EvEraseBkgnd(HDC hdc)
{
  ⋮
}
```

## Child ID notification message macros

The child ID notification message macros provide a number of different ways to handle child ID notification messages. You can

- Handle notification codes from multiple children with a single function
- Pass all notification codes from a child to a response window
- Handle the notification code at the child

Use these macros to facilitate controlling and communicating with child controls. The different types of child ID notification message macros are listed in the following table.

**Table 4.4** Child notification message macros

| Macro | Prototype | Description |
|---|---|---|
| EV_CHILD_NOTIFY(ID, Code, UserName) | **void** *UserName*() | Dispatches the message and notification code to the member function *UserName*. |
| EV_CHILD_NOTIFY_AND_CODE(Id, Code, UserName) | **void** *UserName*(WPARAM code) | Dispatches message *Id* with the notification code *Code* to the function *UserName*. |
| EV_CHILD_NOTIFY_ALL_CODES(Id, UserName) | **void** *UserName*(WPARAM code) | Dispatches message *Id* to the function *UserName*, regardless of the message's notification code. |
| EV_CHILD_NOTIFY_AT_CHILD(Code, UserName) | **void** *UserName*() | Dispatches the notification code *Code* to the child-object member function *UserName*. |

These macros provide different methods for handling child ID notification codes. There are described in the next sections.

### EV_CHILD_NOTIFY

If you want child ID notifications to be handled at the child's parent window, use EV_CHILD_NOTIFY, which passes the notification code as a parameter and lets multiple child ID notifications be handled with a single function. This also prevents having to handle each child's notification message in separate response tables for each control. Instead, each message is handled at the parent, enabling, for example, a dialog box to handle all its controls in its response table.

For example, suppose you have a dialog box called *TTestDialog* that has four buttons. The buttons IDs are ID_BUTTON1, ID_BUTTON2, ID_BUTTON3, and ID_BUTTON4. When the user clicks a button, you want a single function to handle the event, regardless of which button was pressed. If the user double-clicks a button, you want a special function to handle the event. The code would look like this:

```
class TTestDialog : public TDialog
{
  public:
    TTestDialog(TWindow* parent, TResId resId);

    void HandleClick();
    void HandleDblClick1();
    void HandleDblClick2();
    void HandleDblClick3();
    void HandleDblClick4();

  DECLARE_RESPONSE_TABLE(TTestDialog);
};

DEFINE_RESPONSE_TABLE1(TTestDialog, TDialog)
  EV_CHILD_NOTIFY(ID_BUTTON1, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON2, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON3, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON4, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON1, BN_DOUBLECLICKED, HandleDblClick1),
  EV_CHILD_NOTIFY(ID_BUTTON2, BN_DOUBLECLICKED, HandleDblClick2),
  EV_CHILD_NOTIFY(ID_BUTTON3, BN_DOUBLECLICKED, HandleDblClick3),
  EV_CHILD_NOTIFY(ID_BUTTON4, BN_DOUBLECLICKED, HandleDblClick4),
END_RESPONSE_TABLE;
```

## EV_CHILD_NOTIFY_ALL_CODES

If you want all notification codes from the child to be passed to the parent window, use EV_CHILD_NOTIFY_ALL_CODES, the generic handler for child ID notifications. For example, the sample program BUTTONX.CPP defines this response table:

```
DEFINE_RESPONSE_TABLE1(TTestWindow, TWindow)
  EV_COMMAND(ID_BUTTON, HandleButtonMsg),
  EV_COMMAND(ID_CHECKBOX, HandleCheckBoxMsg),
  EV_CHILD_NOTIFY_ALL_CODES(ID_GROUPBOX, HandleGroupBoxMsg),
END_RESPONSE_TABLE;
```

This table handles button, check box, and group box messages. In this case, the parent window (*TTestWindow*) gets all notification messages sent by the child (ID_GROUPBOX). The EV_CHILD_NOTIFY_ALL_CODES macro uses the user-defined function *HandleGroupBoxMsg* to process these messages. As a result, if the user clicks the mouse on one of the group box radio buttons, a message box appears that tells the user which button was selected.

## EV_CHILD_NOTIFY_AND_CODE

You can use the macro EV_CHILD_NOTIFY_AND_CODE if you want the parent window to handle more than one message using the same function. For example:

```
DEFINE_RESPONSE_TABLE1(TTestWindow, TWindow)
  EV_CHILD_NOTIFY_AND_CODE(ID_GROUPBOX, SomeNotifyCode, HandleThisMessage),
  EV_CHILD_NOTIFY_AND_CODE(ID_GROUPBOX, AnotherNotifyCode, HandleThisMessage),
END_RESPONSE_TABLE;
```

If your window has several different messages to handle and uses several different functions to handle these messages, it's better to use EV_CHILD_NOTIFY_AND_CODE instead of EV_CHILD_NOTIFY because EV_CHILD_NOTIFY message-handling function receives no parameters and therefore doesn't know which message it's handling.

## EV_CHILD_NOTIFY_AT_CHILD

To handle child ID notifications at the child window, use EV_CHILD_NOTIFY_AT_CHILD. The sample program NOTITEST.CPP contains the following response table:

```
DEFINE_RESPONSE_TABLE1(TBeepButton, TButton)
  EV_NOTIFY_AT_CHILD(BN_CLICKED, BnClicked),
END_RESPONSE_TABLE;
```

This response table uses the macro EV_NOTIFY_AT_CHILD to tell the child window (*TBeepButton*) to handle the notification code (BN_CLICKED) using the function, *BnClicked*.

Chapter

5

# Command enabling

This chapter discusses the ObjectWindows implementation of command enabling. Most applications provide menu items and control bar or palette button gadgets to access the application's functionality. Some of the commands accessed by these controls are not always available. The menu items and buttons that access these commands should somehow indicate to the application's user when the command isn't available. These menu items and button gadgets can also indicate an application state, such as the current character format, whether a feature is turned on or off, and so on.

ObjectWindows provides a mechanism, known as *command enabling*, that you can use to perform a number of important tasks. This chapter describes how to use ObjectWindows command enabling to

- Turn menu choices and button gadgets on and off
- Set the state of toggled items such as checked menu items and control bar buttons that can be clicked on and off
- Change the text of menu items

For information on menus, please see Chapter 8. For information on button gadgets, such as control bar buttons or palette buttons, and gadget windows, such as control bars and status bars, see Chapter 12.

## Handling command-enabling messages

The basic idea behind ObjectWindows command enabling is that the decision to enable or disable a function should be made by the object that handles the command. ObjectWindows does this by sending the WM_COMMAND_ENABLE message through the same command chain as a WM_COMMAND event. The event is then received by the window that implements the functionality that you are enabling or disabling. The command event chain is discussed in Chapter 4.

When a WM_COMMAND_ENABLE message is sent depends on the type of command item that is affected. *TFrameWindow* performs command enabling for menu items when the user clicks a menu, spawning a WM_INITMENUPOPUP message. Gadget windows perform command enabling for control bar buttons during the window's idle processing.

To handle command-enabling messages for a particular function,

1. Add a member function to the window class to handle the command-enabling message. This function should return **void** and take a single parameter, a reference to a *TCommandEnabler* object. The abstract base class *TCommandEnabler* is declared in the ObjectWindows header file window.h.
2. Place the EV_COMMAND_ENABLE macro in the parent window's response table. This macro takes two parameters, the command identifier and the name of the handler function.

Suppose you have a frame window class that handles a File | Save menu command that uses the command identifier CM_FILESAVE. The class definition would look something like this:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0)
      : TFrameWindow(parent, title), IsDirty(false) {}

  protected:
    void CmFileSave();

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame)
  EV_COMMAND(CM_FILESAVE, CmFileSave),
END_RESPONSE_TABLE;
```

Suppose you don't want the user to be able to access the File | Save command if the file hasn't been modified since it was opened or last saved. Adding a handler function and response table macro to affect the *CmFileSave* function looks something like this:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0)
      : TFrameWindow(parent, title), IsDirty(false) {}

  protected:
    void CmFileSave();

    // This is the command-enabling handler function.
    void CeFileSave(TCommandEnabler& commandEnabler);

  DECLARE_RESPONSE_TABLE(TMyFrame);
```

```
};

DEFINE_RESPONSE_TABLE(TMyFrame)
  EV_COMMAND(CM_FILESAVE, CmFileSave),
  EV_COMMAND_ENABLE(CM_FILESAVE, CeFileSave),
END_RESPONSE_TABLE;
```

Notice that the EV_COMMAND macro and the EV_COMMAND_ENABLE macro both use the same command identifier. Often a single function can be accessed through multiple means. For example, many applications let you open a file through a menu item and also through a button on the control bar. Command enabling in ObjectWindows lets you do command enabling for all means of accessing a function through a single common identifier. The abstraction of command enabling through command-enabling objects saves a great deal of time by removing the need to write multiple command-enabling functions for each different command item.

# Working with command-enabling objects

Once you have received a command-enabling message and the handler function has been called, you can perform a number of actions using the command-enabling object passed to the handler function. This section discusses the various types of ObjectWindows command-enabling objects.

## ObjectWindows command-enabling objects

ObjectWindows provides three predefined command-enabling objects:

- *TCommandEnabler* is the abstract base class for command-enabling objects. It's declared in the ObjectWindows header file window.h.
- *TMenuItemEnabler* is the command-enabling class for menu items. This class enables and disables menu items, sets check marks by menu items, and changes menu item text. This class is declared in the ObjectWindows source file FRAMEWIN.CPP.
- *TButtonGadgetEnabler* is the command-enabling class for button gadgets. This class enables and disables button gadgets and toggles boolean button gadgets. This class is declared in the ObjectWindows source file BUTTONGA.CPP.

## TCommandEnabler: The command-enabling interface

Although in your command-enabling functions you always manipulate an object derived from *TCommandEnabler* as opposed to an actual *TCommandEnabler* object, in practice it appears as if you are working with a *TCommandEnabler* object. *TCommandEnabler* provides a consistent interface for the other command-enabling classes, which implement the appropriate functionality for the type of command object that each class services. Because you never create an instance of the *TMenuItemEnabler* and *TButtonGadgetEnabler* classes, they are declared in source files instead of header files. You don't need to be able to create one of these objects; instead you work with the

basic *TCommandEnabler* interface, while your handler functions are ignorant of the specific command tool that is being handled.

This section describes the *TCommandEnabler* function interface. There are two approaches to the *TCommandEnabler* function interface:

- If you are using existing command-enabling classes, you need to be familiar with the basic interface as implemented in the *TCommandEnabler* class.
- If you are deriving new command-enabling classes, you need to be familiar with the actual implementation of functionality in the *TCommandEnabler* base class.

This section discusses both approaches and points out which aspects are relevant to using existing classes and which are relevant to creating new classes.

## Functions

*TCommandEnabler* has a number of member functions:

- Because *TCommandEnabler* is an abstract class, its constructor is of interest only when you are deriving a new command-enabling class. The *TCommandEnabler* constructor takes two parameters, a uint and an HWND. The uint is the command identifier. The constructor initializes the *Id* data member with the value of the command identifier. The HWND is the handle to the window that received the command-enabling message. The constructor initializes *HWndReceiver* with the value of the HWND parameter.
- *Enable* takes a single bool parameter and returns **void**. The bool parameter indicates whether the command should be enabled or disabled; if it's true, the command is enabled, if it's false, the command is disabled.

  From the standpoint of deriving new classes, all that *TCommandEnabler::Enable* does is perform initialization of data members in the base class. Any other actions required for enabling or disabling a command item must be handled in the derived class. For example, *TMenuItemEnabler* performs all the work necessary to turn menu items on or off. Derived classes' *Enable* functions should always call *TCommandEnabler::Enable*.
- *SetText* takes a single parameter, a **const char far***, and returns **void**. This function sets the text of the command item to the string passed in the character array parameter. *SetText* has no effect on button gadgets.

  *SetText* is declared as a pure virtual; you *must* declare and define *SetText* in classes derived from *TCommandEnabler*. Whatever steps are needed to implement this functionality in your command item must be done in the derived *SetText* function. If, as is the case in *TButtonGadgetEnabler*, there is no valid application for the *SetText* function, you can simply implement it as an empty function.
- *SetCheck* takes a single **int** parameter and returns **void**. This function toggles the command item on or off, depending on the value of the **int** parameter. This parameter can be one of three enumerated values defined in the *TCommandEnabler* class, *Unchecked*, *Checked*, or *Indeterminate*. *Unchecked* sets the state of the command item to be unchecked, *Checked* sets the state of the command item to be checked, and *Indeterminate* sets the command item to its indeterminate state. The nature of the indeterminate state is defined by the command item:

- For menu items, the indeterminate state is the same as unchecked.
- For button gadgets, the indeterminate state is an intermediate state between checked and unchecked.

*SetCheck* is declared as a pure virtual; you *must* declare and define *SetCheck* in classes derived from *TCommandEnabler*. Whatever steps are needed to implement this functionality in your command item must be done in the derived *SetCheck* function.

- *GetHandled* takes no parameters and returns bool. This function returns true if the command enabler has been handled by calling the *Enable* function. Otherwise, it returns false.
- *IsReceiver* takes a single HWND parameter and returns a bool value. *IsReceiver* returns true if the HWND parameter matches the receiver HWND passed into the *TCommandEnabler* constructor and stored in *HWndReceiver*. Otherwise, it returns false.

### Data members

*TCommandEnabler* contains three data members:

- *Id* is the only public data member. This member contains the identifier for the command. It is declared as a **const** uint and is initialized in the constructor. Once initialized, it cannot be modified.
- *HWndReceiver* contains the handle of the window that implements the command. This is a protected data member and cannot be directly accessed unless you are deriving a class from *TCommandEnabler*. *HWndReceiver* can be accessed indirectly by calling the *IsReceiver* function, which compares the value of the HWND parameter passed in to the value of *HWndReceiver*.
- *Handled* indicates whether the command-enabling object has been dealt with. It is initialized to false in the *TCommandEnabler* constructor and set to true in *TCommandEnabler::Enable*. This is a protected data member and cannot be directly accessed unless you are deriving a class from *TCommandEnabler*. *Handled* can be accessed indirectly by calling the *GetHandled* function, which returns the value of *Handled*.

# Common command-enabling tasks

This section describes how to perform some of the more common tasks for which you'll use command enabling, including

- Enabling and disabling command items
- Changing menu item text
- Toggling command items

## Enabling and disabling command items

Enabling and disabling command items is as simple as calling the *Enable* function in your handler function. You decide the criteria for enabling and disabling a particular

item. For example, if a particular library is not available, you may want to disable any commands that access that library. If your application handles files in a number of different formats, you may want to disable commands that aren't appropriate to the current format.

To enable or disable a command,

1. Add the command-enabling handler function and response table macro to your window class as described on page 49.
2. Define the handler function.
3. Inside the handler function, call the *Enable* member function of the command-enabling object passed into the handler function. The *Enable* function takes a single bool parameter. Call *Enable* with the value of the parameter as true to enable the command, and with the value of the parameter as false to disable the command.

Here's the earlier example class from page 50, but with a bool flag, *IsDirty*, added to tell if the file has been modified since it was opened or last saved, and the *CeFileSave* function added to enable and disable the File | Save command:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0)
      : TFrameWindow(parent, title), IsDirty(false) {}

  protected:
    bool IsDirty;

    void CmFileSave();

    // This is the command-enabling handler function.
    void CeFileSave(TCommandEnabler& commandEnabler);

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame)
  EV_COMMAND(CM_FILESAVE, CmFileSave),
  EV_COMMAND_ENABLE(CM_FILESAVE, CeFileSave),
END_RESPONSE_TABLE;

void
TMyFrame::CeFileSave(TCommandEnabler& ce)
{
  ce.Enable(IsDirty);
}
```

*CeFileSave* checks the *IsDirty* flag. If *IsDirty* is false (the file has not been modified), then disable the *CmFileSave* command by calling *Enable*, passing false as the parameter. If *IsDirty* is true (the file has been modified), then enable the *CmFileSave* command, passing true as the parameter. Because you want to call *Enable* with the true parameter when *IsDirty* is true and vice versa, you can just pass *IsDirty* as the parameter to *Enable*.

This method of enabling and disabling a command works for both menu items and button gadgets. In the preceding example, if you have both a control bar button and a menu item that send the CM_FILESAVE command, both commands are implemented in the *CmFileSave* function. Similarly, command enabling for the control bar button and the menu item is implemented in the *CeFileSave* function.

## Changing menu item text

Changing the text of a menu item is done with the *SetText* function. To change the text of a menu item,

1. Add the command-enabling handler function and response table macro to your window class as described on page 49.
2. Define the handler function.
3. In the handler function, call the *SetText* member function of the command-enabling object passed into the handler function. *SetText* takes a single parameter, a **const far char***. This character array parameter should contain the new text for the menu item. *SetText* returns **void**.

**Note** If you're setting the text for a menu item and turning on a check mark for that menu item in the same function, you must call *SetText* before you call *SetCheck*. Reversing this order removes the check mark. See page 56 for information on setting check marks for menu items.

Suppose your application supports three different file formats, text, binary, and encrypted. You want the File | Save menu item to reflect the format of the file being saved. Here's the example class from earlier on page 50, modified with an **enum** type, *TFormat*, and a *TFormat* data member called *Format*:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0);
    enum TFormat {Text, Binary, Encrypted};

  protected:
    TFormat Format;

    void CmFileSave();

    // This is the command-enabling handler function.
    void CeFileSave(TCommandEnabler& commandEnabler);

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame)
  EV_COMMAND(CM_FILESAVE, CmFileSave),
  EV_COMMAND_ENABLE(CM_FILESAVE, CeFileSave),
END_RESPONSE_TABLE;
```

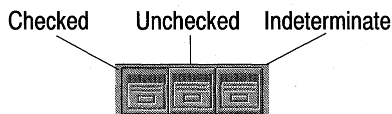
```
void
TMyFrame::CeFileSave(TCommandEnabler& ce)
{
  switch(Format) {
    case Text:
      ce.SetText("Save as text file");
      break;
    case Binary:
      ce.SetText("Save as binary file");
      break;
    case Encrypted:
      ce.SetText("Save as encrypted file");
      break;
    default:
      ce.SetText("Save");
  }
}
```

## Toggling command items

You can use command item toggling to provide the users of your applications visual cues about what functions are enabled, various application states, and so on. Anything that can be presented in a boolean fashion, such as on and off, in and out, and so on, can be represented by command item toggling.

There are two different types of toggling implemented in ObjectWindows, but both are implemented the same way. You can turn check marks by menu items on and off. You can also "check" and "uncheck" button gadgets so that the gadget stands out when it's off and is recessed and light when it's on. There is also a third *indeterminate* state that indicates when something is not checked or unchecked. The meaning of this state is mostly up to you, but usually indicates a situation where the criteria for being enabled or disabled is mixed. For example, many word processors have control bar buttons that indicate the current text format, such as a button with a "B" on it to indicate bold text. This button is unchecked when the current text format is not bold, and checked when the format is bold. But if a block of text contains text, some of which is bold and some not, the button is placed in its indeterminate state. Figure 5.1 shows a button gadget in each of the three states:

**Figure 5.1** Button gadget states




A variation of toggling button gadgets is that you can enable or disable an exclusive button gadget. Exclusive button gadgets function just like radio buttons. In a group of exclusive button gadgets only one button gadget can be on at a time. Enabling another button gadget in the group disables the previously enabled button gadget.

To toggle a command item,

1. Add the command-enabling handler function and response table macro to your window class as described on page 49.
2. Define the handler function.
3. Inside the handler function, call the *SetCheck* member function of the command-enabling object passed into the handler function. The *SetCheck* function takes a single **int** parameter. Call *SetCheck* with one of the enumerated values defined in *TCommandEnabler*: *Checked*, *Unchecked*, or *Indeterminate*.

**Note** If you are turning on a check mark for a menu item and setting the text for that menu item in the same function, you must call *SetText* before you call *SetCheck*. Reversing this order removes the check mark. See page 56 for information on setting check marks for menu items.

A common use for toggling command items is to let the user of your application specify whether some feature should be active. For example, suppose your application provides both a menu item and control bar button to access the *CmFileSave* function. Many applications provide "fly-over" hints, short descriptions that appear in the status bar when the pointer moves over a menu item or button gadget. You may want to let the user turn these hints off. To provide this option to the user,

1. Add a new command identifier to your application, such as CM_TOGGLEHINTS.
2. Add a new menu, perhaps named Options, with a menu item Fly-over Hints.
3. You can also add a new button to your button bar (see Chapter 12 for information on adding a new button gadget to your control bar).
4. Add a function to handle the CM_TOGGLEHINTS event and actually turn the hints on and off.
5. Add a command-enabling function to check and uncheck the command items.

Here's the example class from earlier on page 50, modified to use a decorated frame window. The user can toggle hints by choosing the command item set up for this.

```
class TMyDecFrame : public TDecoratedFrame
{
  public:
    TMyDecFrame(TWindow *parent = 0, char *title = 0, TWindow* client)
      : TDecoratedFrame(parent, title, client), hintMode (true) {}

    // Cb must be set by the application object during the InitMainWindow function.
    TControlBar* Cb;

  protected:
    // hintMode indicates whether the hints are currently on or off.
    bool HintMode;

    // This is the function that actually turns the hints on and off.
    void CmToggleHints();

    // This is the command-enabling handler function.
    void CeToggleHints(TCommandEnabler& commandEnabler);
```

```
  DECLARE_RESPONSE_TABLE(TMyDecFrame);
};

DEFINE_RESPONSE_TABLE(TMyDecFrame)
  EV_COMMAND(CM_TOGGLEHINTS, CmToggleHints),
  EV_COMMAND_ENABLE(CM_TOGGLEHINTS, CeToggleHints),
END_RESPONSE_TABLE;

void
TMyDecFrame::CmToggleHints()
{
  if(HintMode)
    Cb->SetHintMode(TGadgetWindow::EnterHints);
  else
    Cb->SetHintMode(TGadgetWindow::NoHints);
  HintMode = !HintMode;
}

void
TMyDecFrame::CeToggleHints(TCommandEnabler& ce)
{
  ce.SetChecked(HintMode);
}
```

Note that the control bar is set up by the application object in its *InitMainWindow* function. The code for this is not shown here. For an explanation of application objects and the *InitMainWindow* function see Chapter 2. For an explanation of button gadgets and control bars, see Chapter 12. For a working example of command item toggling, see the example EXAMPLES/OWL/OWLAPPS/MDIFILE.

Chapter

6

# ObjectWindows exception handling

ObjectWindows provides a robust exception-handling mechanism for dealing with exceptional situations. An exceptional situation is any situation that falls outside of your application's normal operating parameters. This can be something as innocuous as an unexpected user response or something as serious as an invalid handle or memory allocation failure. Exception handling provides a clean, efficient way to deal with these and other conditions.

This chapter describes the ObjectWindows exception-handling encapsulation, including

- Exception class hierarchy
- Exception resource identifiers
- Code macros, which make it easy to turn exception handling off and on

You should be thoroughly familiar with C++ exception handling before reading this chapter. C++ exception handling is described in Chapter 4 of the *Borland C++ Programmer's Guide.*

## ObjectWindows exception hierarchy

ObjectWindows provides a number of classes that can be thrown as exceptions. Based on the *TXBase* and *TXOwl* classes, these exception classes can inform the user of the existing exceptional state, prompt the user for a course of action, create new exception objects, throw exceptions, and so on. There are four exception classes that are implemented as independent classes:

- *TXBase* is the base class for all ObjectWindows and ObjectComponents exception classes. *TXBase* is derived from the Borland C++ *xmsg* class. *xmsg* is described in Chapter 4 of the *Borland C++ Programmer's Guide.*

- *TXOwl* is derived from *TXBase*. *TXOwl* is the base class for the ObjectWindows exception classes.

- *TXCompatibility* describes exceptions that occur when *TModule::Status* is non-zero. This provides backwards compatibility between the ObjectWindows 1.0 method of detecting exceptional situations and the ObjectWindows 2.*x* exception hierarchy. *TXCompatibility* maps the value of *TModule::Status* to a resource string identifier.
- *TXOutOfMemory* describes an exception that occurs when an attempt to allocate memory space for an object fails. This is analogous to the *xalloc* object thrown when **new** fails to properly allocate memory.

Two other classes, *TXOle* and *TXAuto*, are derived from *TXBase*. These classes provide exception handling for the ObjectComponents classes. They are described in the *ObjectWindows Reference Guide*.

## Working with TXBase

As the base class for the ObjectWindows exception classes, *TXBase* provides the basic interface for working with ObjectWindows exceptions. *TXBase* can perform a number of functions:

- It can construct itself, initializing its base *xmsg* object.
- It can clone itself, making a copy of the exception object.
- It can throw itself as an exception object.

### Constructing and destroying TXBase

*TXBase* provides two public constructors:

```
TXBase(const string& msg);
TXBase(const TXBase& src);
```

The first constructor initializes the *xmsg* base class with the value of the *string* parameter, calling the *xmsg* constructor that takes a *string* parameter. The second creates a new object that is a copy of the *TXBase* object passed in as a parameter.

Both constructors increment the *TXBase* data member *InstanceCount*. *InstanceCount* is a **static int**, meaning there is only a single instance of the member no matter how many actual *TXBase* or *TXBase*-derived objects exist in the application. The *TXBase* destructor decrements *InstanceCount*. The destructor is declared virtual to allow easy overriding of the destructor.

Because each new *TXBase* or *TXBase*-derived object increments *InstanceCount*, and each deleted *TXBase* or *TXBase*-derived object decrements *InstanceCount*, the value of *InstanceCount* reflects the total number of *TXBase* and *TXBase*-derived objects existing in the application at the time. To access *InstanceCount* from outside a *TXBase* or *TXBase*-derived class, qualify the name *InstanceCount* with a *TXBase::* scope qualifier.

### Cloning exception objects

*TXBase* contains a function called *Clone*. This function takes no parameters and returns a *TXBase**. *Clone* creates a copy of the current exception object by allocating a new *TXBase* object with **new** and passing a dereferenced **this** pointer to the copy constructor.

```
TXBase*
TXBase::Clone()
```

```
{
  return new TXBase(*this);
}
```

It is important to note that any classes derived from *TXBase* must override this function to use the proper constructor. For example, the *TXOwl* class, which is derived from *TXBase*, implements the *Clone* function like this:

```
TXOwl*
TXOwl::Clone()
{
  return new TXOwl(*this);
}
```

### Throwing TXBase exceptions

Once you have a *TXBase* object, either by creating it or cloning it, you can throw the object one of three ways.

- Use the **throw** keyword followed by the object name

  ```
  TXBase xobj("Some exception...");
  throw xobj;
  ```

- Use the ObjectWindows THROW macro, which corresponds to the C++ keyword **throw**. See page 63 for an explanation of the ObjectWindows exception-handling macros. The previous example would look like this:

  ```
  TXBase xobj("Some exception...");
  throw xobj;
  ```

- Call the exception object's *Throw* function:

  ```
  TXBase xobj("Some exception...");
  xobj.Throw();
  ```

  This method provides for strict type safety when you throw the exception. It also provides a polymorphic interface when throwing the exception, so that the function that catches a *TXBase*-derived exception object can treat the object as a *TXBase*, regardless of what it actually is.

## Working with TXOwl

As the base class for the ObjectWindows exception classes, *TXOwl* provides the basic interface for working with ObjectWindows exceptions. In addition to the functionality provided in the *TXBase* class, *TXOwl* can perform a number of other functions.

- It can construct itself, initializing its base objects.
- It can clone itself, making a copy of the exception object.
- It can pass unhandled exceptions to the application object's *Error* function or to the global exception handler *HandleGlobalException* (*HandleGlobalException* is discussed later).

### Constructing and destroying TXOwl

*TXOwl* has two constructors to provide flexibility in passing the exception message string:

```
TXOwl(const string& str, unsigned resId = 0);
TXOwl(unsigned resId, TModule* module = ::Module);
```

The first constructor initializes the *TXBase* base object with the value of the *string* parameter. The **unsigned** parameter is used as an error number.

The second constructor loads the string resource identified by *resId* and uses the string to initialize *TXBase*. The *TModule** identifies the module from which the resource should be loaded. It defaults to the global current module pointer *Module,* meaning the resource should be loaded from the current module or application.

The *TXOwl* destructor has no default functionality other than that inherited from *TXBase*.

### Cloning TXOwl and TXOwl-derived exception objects

*TXOwl* also contains the *Clone* function. This function takes no parameters and returns a *TXOwl**. *Clone* creates a copy of the current exception object by allocating a new *TXOwl* object with **new** and passing a dereferenced **this** pointer to the automatic copy constructor.

```
TXOwl*
TXOwl::Clone()
{
  return new TXOwl(*this);
}
```

It is important to note that any classes derived from *TXOwl* must override this function to use the proper constructor. For example, the *TXOutOfMemory* class, which is derived from *TXOwl,* implements the *Clone* function like this:

```
TXOwl*
TXOutOfMemory::Clone()
{
  return new TXOutOfMemory(*this);
}
```

Note that the return type is still *TXOwl**. This lets the ObjectWindows exception-handling functions treat any exception object as a *TXOwl* object, in keeping with the polymorphic nature of the ObjectWindows hierarchy. But also note that the return type for *TXOwl::Close* differs from the *TXBase::Clone* function. That is because, while *TXBase* provides the basic functionality for the ObjectWindows and ObjectComponents exception classes, *TXOwl* provides the basic interface for the ObjectWindows exception classes.

## Specialized ObjectWindows exception classes

A number of regular ObjectWindows classes implement specialized exception classes, all of which are based on *TXOwl* but are defined within the implementing class

definition to provide name scoping. The following table describes these classes, along with the unique functionality of each class. The various IDS_* resources mentioned in the table, along with many others, are described in Chapter 2 of the *ObjectWindows Reference Guide.*

**Table 6.1** Specialized exception classes

| Parent class | Exception class | Function |
|---|---|---|
| *TApplication* | *TXInvalidMainWindow* | Initializes the exception message with the IDS_INVALIDMAINWINDOW string resource. This object is thrown when the *MainWindow* member of *TApplication contains either an invalid pointer or a pointer to an invalid window.* |
| *TModule* | *TXInvalidModule* | Initializes the exception message with the IDS_INVALIDMODULE string resource. This exception is thrown in the *TModule* constructor when the module's *HInstance* is invalid. |
| *TWindow* | *TXWindow* | Initializes the exception message with the window title and with a string resource passed to the *TXWindow* constructor. This exception is thrown in situations where an error relating to a window object has occurred. |
| *TMenu* | *TXMenu* | Initializes the exception message with a string resource passed to the *TXMenu* constructor. By default this is the IDS_GDIFAILURE string resource. This exception is thrown when a menu object's handle is invalid. |
| *TValidator* | *TXValidator* | Initializes the exception message with a string resource passed to the *TXValidator* constructor. By default this is the IDS_VALIDATORSYNTAX string resource. This exception is thrown when a validator expression is corrupt or invalid. |
| *TGdiBase* | *TXGdi* | Initializes the exception message with a string resource passed to the *TXGdi* constructor, along with the GDI object handle. By default, the string resource is IDS_GDIFAILURE and the GDI object handle is 0. This exception is thrown in numerous situations when an error relating to a graphics object has occurred. |
| *TPrinter* | *TXPrinter* | Initializes the exception message with a string resource passed to the *TXPrinter* constructor. By default this is the IDS_PRINTERERROR string resource. This exception is thrown when the printer's device context is invalid. |

# ObjectWindows exception-handling macros

ObjectWindows provides a number of macros for implementing exception handling. Although you can use the standard C++ keywords such as **try**, **catch**, **throw**, and so on, the ObjectWindows macros enable you to turn exception handling on and off simply by defining or not defining a single symbol. The macros provided are

- TRY
- THROW(x)
- THROWX(x)
- RETHROW
- CATCH(x)

These macros are explained later in this section.

## Turning ObjectWindows exceptions on and off

The symbol that switches exception handling on and off in ObjectWindows applications is NO_CPP_EXCEPTIONS. The value (or lack of value) assigned to NO_CPP_EXCEPTIONS doesn't matter. What matters is whether it is defined. If it's not, the exception-handling macros expand to implement exception handling. If it is defined, the macros provide only the barest functionality by aborting the application when an exception is thrown. The precise behaviors of the macros when exception handling is switched on and off is described later.

There are many different methods for defining NO_CPP_EXCEPTIONS. This list doesn't contain all the ways to define it, but makes a few suggestions.

- You can specify the `-DNO_CPP_EXCEPTIONS` option on the MAKE command line. This defines the macro, but with no specific value.
- You can define a symbol using a graphical development environments such as the Borland C++ IDE. Use the method provided in your graphical development to define the NO_CPP_EXCEPTIONS symbol.
- You can define NO_CPP_EXCEPTIONS in your source code. This is a less desirable method than the previous ones, mainly because if you're using some type of MAKE or dependency-checking program for building your application, modifying the source code modifies the time stamp on the file. You might or might not want the time stamp to change.

## Macro expansion

The exception-handling macros in ObjectWindows behave differently depending on whether NO_CPP_EXCEPTIONS is defined. The following table explains how each macro is expanded depending on the state of NO_CPP_EXCEPTIONS:

**Table 6.2** ObjectWindows exception-handling macro expansion

| Macro | NO_CPP_EXCEPTIONS defined | NO_CPP_EXCEPTIONS not defined |
|---|---|---|
| TRY | Expands to nothing, removing the **try** statement. | Expands to **try**, allowing the code in the **try** block to be tested for thrown exceptions. |
| THROW(x) | Calls the *abort* function. | Expands to **throw**(*x*), throwing the *x* object if the exceptional conditions are met. |
| THROWX(x) | Calls the *abort* function. | Expands to *x*.Throw(), calling the object *x*'s *Throw* function. This macro should only be used with *TXBase*-derived classes. |
| RETHROW | Expands to nothing, removing the **throw** statement. | Expands to **throw**. This macro should be used only inside of **catch** (or CATCH) clauses to rethrow the caught exception. |
| CATCH(x) | Expands to nothing, removing the **catch** statement. | Expands to **catch** x, catching exceptions thrown with objects of type *x*. |

Chapter

# 7

# Window objects

ObjectWindows window objects provide an interface wrapper around windows, making dealing with windows and their children and controls much easier. ObjectWindows provides several different types of window objects:

- Layout windows (described starting on page 69)
- Frame windows (described starting on page 75)
- Decorated frame windows (described starting on page 78)
- MDI windows (described starting on page 80)

Another class of window objects, called gadget windows, is discussed in Chapter 12.

## Using window objects

This section explains how to create, display, and fill window objects. It describes how to perform the following tasks:

- Constructing window objects
- Setting creation attributes
- Creating window interface elements

The different types of windows discussed in this chapter—frame windows, layout windows, decorated frame windows, and MDI windows—are all examples of window objects. The information in this section applies to all the different types of window objects.

### Constructing window objects

Window objects represent interface elements. The object is connected to the element through a handle stored in the object's *HWindow* data member. *HWindow* is inherited from *TWindow*. When you construct a window object, its interface element doesn't yet

exist. You must create it in a separate step. *TWindow* also has a constructor that you can use in a DLL to create a window object for an interface element that already exists.

### Constructing window objects with virtual bases

Several ObjectWindows classes use *TWindow* or *TFrameWindow* as a virtual base. These classes are *TDialog*, *TMDIFrame*, *TTinyCaption*, *TMDIChild*, *TDecoratedFrame*, *TLayoutWindow*, *TClipboardViewer*, *TKeyboardModeTracker*, and *TFrameWindow*. In C++, virtual base classes are constructed first, which means that the derived class' constructor cannot specify default arguments for the base class constructor. There are two ways to handle this problem:

- Explicitly construct your immediate base class or classes and any virtual base classes when you construct your derived class.
- Use the virtual base's default constructor. Both *TWindow* and *TFrameWindow* have a default constructor. They also each have an *Init* function that lets you specify parameters for the base class; call this *Init* function in the constructor of your derived class to set any parameters you need in the base class.

Here's a couple of examples showing how to construct a window object using the each of the methods described above:

```
class TMyWin : public TFrameWindow
{
  public:
    // This constructor calls the base class constructors
    TMyWin(TWindow *parent, char *title)
      : TFrameWindow(parent, title),
        TWindow(parent, title) {}
}

TMyWin *myWin = new TMyWin(GetMainWindow(), "Child window");

class TNewWin : virtual public TWindow
{
  public:
    TNewWin(TWindow *parent, char *title);
}

TNewWin::TNewWin(TWindow *parent, char *title)
{
  // This constructor uses the default base class constructors and calls Init
  Init(parent, title, IDL_DEFAULT);
};

TNewWin *newWin = new TMyWin(GetMainWindow(), "Child window");
```

## Setting creation attributes

A typical Windows application has many different types of windows: overlapped or pop-up, bordered, scrollable, and captioned, to name a few. The different types are

selected with *style attributes*. Style attributes, as well as a window's title, are set during a window object's initialization and are used during the interface element's creation.

A window object's creation attributes, such as style and title, are stored in the object's *Attr* member, a *TWindowAttr* structure. Table 7.1 shows *TWindowAttr*'s members.

**Table 7.1** Window creation attributes

| Member | Type | Description |
|---|---|---|
| Style | *uint32* | Style constant. |
| ExStyle | *uint32* | Extended style constant. |
| X | **int** | The horizontal screen coordinate of the window's upper-left corner. |
| Y | **int** | The vertical screen coordinate of the window's upper-left corner. |
| W | **int** | The window's initial width in screen coordinates. |
| H | **int** | The window's initial height in screen coordinates. |
| Menu | *TResId* | ID of the window's menu resource. You should not try to directly assign a menu identifier to *Attr.Menu*! Use the *AssignMenu* function instead. |
| Id | **int** | Child window ID for communicating between a control and its parent. *Id* should be unique for all child windows of the same parent. If the control is defined in a resource, its *Id* should be the same as the resource ID. A window should never have both *Menu* and *Id* set, since these members actually occupy the same in the window's HWND structure. |
| Param | **char far *** | Used by *TMDIClient* to hold information about the MDI frame and child windows. |
| AccelTable | *TResId* | ID of the window's accelerator table resource. |

## Overriding default attributes

Table 7.2 lists the default window creation attributes. You can override those defaults in a derived window class' constructor by changing the values in the *Attr* structure. For example:

```
TTestWindow::TTestWindow(TWindow* parent, const char* title)
  : TFrameWindow(parent, title),
    TWindow(parent, title)
{
  Attr.Style &= (WS_SYSMENU | WS_MAXIMIZEBOX);
  Attr.Style |= WS_MINIMIZEBOX;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 415;
  Attr.H = 355;
}
```

## Child-window attributes

You can set the attributes of a child window in the child window's constructor or in the code that creates the child window. When you change the attributes in the parent window object's constructor, you need to use a pointer to the child window object to get access to its *Attr* member.

```
TTestWindow::TTestWindow(TWindow* parent, const char* title)
  : TWindow(parent, title)
```

```
{
  TWindow helpWindow(this, "Help System");

  helpWindow.Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
  helpWindow.Attr.X = 100;
  helpWindow.Attr.Y = 100;
  helpWindow.Attr.W = 300;
  helpWindow.Attr.H = 300;
  helpWindow.SetCursor(0, IDC_HAND);
}
```

Table 7.2 shows some default values you might want to override for *Attr* members. A default value of 0 means to use the Windows default value.

**Table 7.2** Default window attributes

| *Attr* member | Default value |
|---|---|
| *Style* | WS_CHILD \| WS_VISIBLE |
| *ExStyle* | 0 |
| *X* | 0 |
| *Y* | 0 |
| *W* | 0 |
| *H* | 0 |
| *Menu* | 0 |
| *Id* | 0 |
| *Param* | 0 |
| *AccelTable* | 0 |

## Creating window interface elements

Once you've constructed a window object, you need to tell Windows to create the associated interface element. Do this by calling the object's *Create* member function:

```
window.Create();
```

*Create* does the following things:

- Creates the interface element
- Sets *HWindow* to the handle of the interface element
- Sets members of *Attr* to the actual state of the interface element (*Style, ExStyle, X, Y, H, W*)
- Calls *SetupWindow*

An application's main window is automatically created by *TApplication::InitInstance*. You don't need to call *Create* yourself to create the main window. See page 23 for more information about main windows.

Two ObjectWindows exceptions can be thrown while creating a window object's interface element. You should therefore enclose calls to *Create* within a **try/catch** block to handle any memory or resource problems your application might encounter. *Create*

throws a *TXInvalidWindow* exception when the window can't be created. *SetupWindow* throws *TXInvalidChildWindow* when a child window in the window can't be created. Both exceptions are usually caused by insufficient memory or other resources. Here is an example of using exceptions to catch an error while creating a window object:

```
try
{
  TWindow* window = new TMyWindow(this);
  window->Create();
}
catch(TXOwl& exp)
{
  MessageBox(exp.why.c_str(), "Window creation error");
throw(exp);
}
```

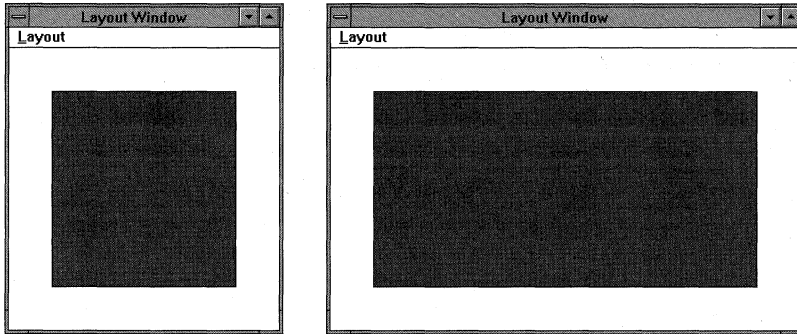
ObjectWindows exception objects are described in Chapter 6.

# Layout windows

This section discusses layout windows. Layout windows are encapsulated in the class *TLayoutWindow*, which is derived from *TWindow*. Along with *TFrameWindow*, *TLayoutWindow* provides the basis for decorated frame windows and their ability to arrange decorations in the frame area.

Layout windows are so named because they can lay out child windows in the layout window's client area. The children's locations are determined relative to the layout window or another child window (known as a *sibling*). The location of a child window depends on that window's *layout metrics*, which consist of a number of rules that describe the window's X and Y coordinates, its height, and its width. These rules are usually based on a sibling window's coordinates and, ultimately, on the size and arrangement of the layout window. Figure 7.1 shows two shots of an example layout window with a child window in the client area. In this example, the child's layout metrics specify that the child is to remain the same distance from each side of the layout window. Notice how, in the first shot, the child window is rather small. Then, in the second shot, the layout window has been enlarged. The child window, following its layout constraints, got larger so that each of its edges stayed the same distance from the edge of the layout window.

**Figure 7.1** Example layout windows




Layout metrics for a child window are contained in a class called *TLayoutMetrics*. A layout metrics object consists of a number of *layout constraints*. Each layout constraint describes a rule for finding a particular dimension, such as the X coordinate or the width of the window. It takes four layout constraints to fully describe a layout metrics object. Layout constraints are contained in a structure named *TLayoutConstraints*, but you usually use one of the *TLayoutConstraints*-derived classes, such as *TEdgeConstraint*, *TEdgeOrWidthConstraint*, or *TEdgeOrHeightConstraint*.

## Layout constraints

Layout constraints specify a relationship between an edge or dimension of one window and an edge or dimension of a sibling window or the parent layout window. This relationship can be quite flexible. For example, you can set the width of a window to be a percentage of the width of the parent window, so that whenever the parent is resized, the child window is resized to take up the same relative window area. You can also set the left edge of a window to be the same as the right edge of another child, so that when the windows are moved around, they are tied together. You can even constrain a window to occupy an absolute size and position in the client area.

The three types of constraints most often used are *TEdgeConstraint*, *TEdgeOrWidthConstraint*, and *TEdgeOrHeightConstraint*. These structures constitute the full set of constraints used in the *TLayoutMetrics* class. *TEdgeOrWidthConstraint* and *TEdgeOrHeightConstraint* are derived from *TEdgeConstraint*. From the outside, these three objects look almost the same. When this section discusses *TEdgeConstraint*, it is referring to all three objects—*TEdgeConstraint*, *TEdgeOrWidthConstraint*, and *TEdgeOrHeightConstraint*—unless the other two classes are explicitly excluded from the statement.

### Defining constraints

The most basic way to define a constraining relationship (that is, setting up a relationship between an edge or size of one window and an edge or size of another window) is to use the *Set* function. The *Set* function is defined in the *TEdgeConstraint* class and subsequently inherited by *TEdgeOrWidthConstraint* and *TEdgeOrHeightConstraint*.

Here is the *Set* function declaration:

```
void Set(TEdge edge, TRelationship rel,
        TWindow* otherWin, TEdge otherEdge,
        int value = 0);
```

where:

- *edge* specifies which part of the window you are constraining. For this, there is the **enum** *TEdge*, which has five possible values:
  - *lmLeft* specifies the left edge of the window.
  - *lmTop* specifies the top edge of the window.
  - *lmRight* specifies the right edge of the window.
  - *lmBottom* specifies the bottom edge of the window.
  - *lmCenter* specifies the center of the window. The object that owns the constraint, such as *TLayoutMetrics*, decides whether this means the vertical center or the horizontal center.
- You can also specify the window's width or height as a constraint, but only with *TEdgeOrWidthConstraint* and *TEdgeOrHeightConstraint*. For this, there is the **enum** *TWidthHeight*. *TWidthHeight* has two possible values:
  - *lmWidth* specifies that the width of the window should be constrained.
  - *lmHeight* specifies that the height of the window should be constrained.
- *rel* specifies the relationship between the two edges:

**Table 7.3** Default window attributes

| *rel* | Relationship |
|---|---|
| lmAsIs | This dimension is constrained to its current value. |
| lmPercent Of | This dimension is constrained to a percentage of the constraining edge's size. This is usually used with a constraining width or height. |
| lmAbove | This dimension is constrained to a certain distance above its constraining edge. |
| lmLeftOf | This dimension is constrained to a certain distance to the left of its constraining edge. |
| lmBelow | This dimension is constrained to a certain distance below its constraining edge. |
| lmRightOf | This dimension is constrained to a certain distance to the right of its constraining edge. |
| lmSameAs | This dimension is constrained to the same value as its constraining edge. |
| lmAbsolute | This dimension is constrained to an absolute coordinate or size. |

- *otherWin* specifies the window with which you are constraining your child window. You must use the value *lmParent* when specifying the parent window. Otherwise, pass a pointer to the *TWindow* or *TWindow*-derived object containing the other window.
- *otherEdge* specifies the particular edge of *otherWin* with which you are constraining your child window. *otherEdge* can have any of the same values that are allowed for *edge*.

- *value* means different things, depending on the value of *rel*:

| *rel* | Meaning of *value* |
|---|---|
| *lmAsIs* | *value* has no meaning and should be set to 0. |
| *lmPercentOf* | *value* indicates what percent of the constraing measure the constrained measure should be. |
| *lmAbove* | *value* indicates how many units above the constraining edge the constrained edge should be. |
| *lmLeftOf* | *value* indicates how many units to the lef t of the constraining edge the constrained edge should be. |
| *lmBelow* | *value* indicates how many units below the constraining edge the constrained edge should be. |
| *lmRightOf* | *value* indicates how many units to the right of the constraining edge the constrained edge should be. |
| *lmSameAs* | *value* has no meaning and should be set to 0. |
| *lmAbsolute* | *value* is the absolute measure for the constrained edge:<br>When *edge* is *lmLeft, lmRight,* or sometimes *lmCenter, value* is the X coordinate for the edge.<br>When *edge* is *lmTop, lmBottom,* or sometimes *lmCenter, value* is the Y coordinate for the edge.<br>When *edge* is *lmWidth* or *lmHeight, value* represents the size of the constraint.<br>The owning object determines whether *lmCenter* represents an X or Y coordinate. See page 70. |

- The meaning of *value* is also dependent on the value of *Units*. *Units* is a *TMeasurementUnits* member of *TLayoutConstraint*. *TMeasurementUnits* is an **enum** that describes the type of unit represented by *value*. *Units* can be either *lmPixels* or *lmLayoutUnits*. *lmPixels* indicates that *value* is meant to represent an absolute number of physical pixels. *lmLayoutUnits* indicates that *value* is meant to represent a number of logical units. These layout units are based on the size of the current font of the layout window.

*TEdgeConstraint* also contains a number of functions that you can use to set up predefined relationships. These correspond closely to the relationships you can specify in the *Set* function. In fact, these functions call *Set* to define the constraining relationship. You can use these functions to set up a majority of the constraint relationships you define.

The following four functions work in a similar way:

```
void LeftOf(TWindow* sibling, int margin = 0);
void RightOf(TWindow* sibling, int margin = 0);
void Above(TWindow* sibling, int margin = 0);
void Below(TWindow* sibling, int margin = 0);
```

Each of these functions place the child window in a certain relationship with the constraining window *sibling*. The edges are predefined, with the constrained edge being the opposite of the function name and the constraining edge being the same as the function name.

For example, the *LeftOf* function places the child window to the left of *sibling*. This means the constrained edge of the child window is *lmRight* and the constraining edge of *sibling* is *lmLeft*.

You can set an edge of your child window to an absolute value with the *Absolute* function:

```
void Absolute(TEdge edge, int value);
```

*edge* indicates which edge you want to constrain, and *value* has the same value as when used in *Set* with the *lmAbsolute* relationship.

There are two other shortcut functions you can use:

```
void SameAs(TWindow* otherWin, TEdge edge);
void PercentOf(TWindow* otherWin, TEdge edge, int percent);
```

These two use the same edge for the constrained window and the constraining window; that is, if you specify *lmLeft* for *edge*, the left edge of your child window is constrained to the left edge of *otherWin*.

## Defining constraining relationships

A single layout constraint is not enough to lay out a window. For example, specifying that one window must be 10 pixels below another window doesn't tell you anything about the width or height of the window, the location of the left or right borders, or the location of the bottom border. It only tells you that one edge is located 10 pixels below another window.

A combination of layout constraints can define fully a window's location (there are some exceptions, as discussed on page 74). The class *TLayoutMetrics* uses four layout constraint structures—two *TEdgeConstraint* objects named *X* and *Y*, a *TEdgeOrWidthConstraint* named *Width*, and a *TEdgeOrHeightConstraint* named *Height*.

*TLayoutMetrics* is a fairly simple class. The constructor takes no parameters. The only thing it does is to set up each layout constraint member. For each layout constraint, the constructor

- Zeroes out the value for the constraining window.
- Sets the constraint's relationship to lmAsIs.
- Sets units to *lmLayoutUnits*.
- Sets the value to 0.

The only difference is to *MyEdge*, which indicates to which edge of the window this constraint applies. *X* is set to *lmLeft*, *Y* is set to *lmTop*, *Width* is set to *lmWidth*, and *Height* is set to *lmHeight*.

Once you have constructed a *TLayoutMetrics* object, you need to set the layout constraints for the window you want to lay out. You can use the functions described in the preceding section for setting each layout constraint.

It is important to realize that the labels *X*, *Y*, *Width*, and *Height* are more labels of convenience than strict rules on how the constraints should be used. *X* can represent the X coordinate of the left edge, the right edge, or the center. You can combine this with the *Width* constraint—which can be one of *lmCenter*, *lmRight*, or *lmWidth*—to completely

define the window's X-axis location and width. Using all of the edge constraints is easy, and is useful in situations where tiling is performed.

The simplest way is to assign an X coordinate to X and a width to width. But you could also set the edge for X to *lmCenter* and the edge for *Width* to *lmRight*. So *Width* doesn't really represent a width, but the X-coordinate of the window's right edge. If you know the X-coordinate of the right edge and the center, it's easy to calculate the X-coordinate of the left edge.

To better understand how constraints work together to describe a window, try building and running the example application LAYOUT in the directory EXAMPLES\OWL\OWLAPI\LAYOUT. This application has a number of child windows in a layout window. A dialog box you can access from the menu lets you change the constraints of each of the windows and then see the results as the windows are laid out. Be careful, though. If you specify a set of layout constraints that doesn't fully describe a window, the application will probably crash, or, if diagnostics are on, a check will occur. The reason for this is discussed in the next section.

### Indeterminate constraints

You must be careful about how you specify your layout constraints. The constraints available in the *TLayoutMetrics* class give you the ability to fully describe a window. But they do not guarantee that the constraints you use will fully describe a window. In cases where the constraints do not fully describe a window, the most likely result is an application crash.

## Using layout windows

Once you've set up layout constraints, you're ready to create a layout window to contain your child windows. Here's the constructor for *TLayoutWindow*:

```
TLayoutWindow(TWindow* parent,
              const char far* title = 0,
              TModule* module = 0);
```

where:

- *parent* is the layout window's parent window.
- *title* is the layout window's title. This parameter defaults to a null string.
- *module* is passed to the *TWindow* base class constructor as the *TModule* parameter for that constructor. This parameter defaults to 0.

After the layout window is constructed and displayed, there are a number of functions you can call:

- The *Layout* function returns **void** and takes no parameters. This function tells the layout window to look at all its child windows and lay them out again. You can call this to force the window to recalculate the boundaries and locations of each child window. You usually want to call *Layout* after you've moved a child window, resized the layout window, or anything else that could affect the constraints of the child windows.

Note that *TLayoutWindow* overrides the *TWindow* version of *EvSize* to call *Layout* automatically whenever a WM_SIZE event is caught. If you override this function yourself, you should be sure either to call the base class version of the function or call *Layout* in your derived version.

- *SetChildLayoutMetrics* returns **void** and takes a *TWindow* **&** and a *TLayoutMetrics* **&** as parameters. Use this function to associate a set of constraints contained in a *TLayoutMetrics* object with a child window. Here is an example of creating a *TLayoutMetrics* object and associating it with a child window:

```
TMyLayoutWindow::TMyLayoutWindow(TWindow* parent, char far* title)
  : TLayoutWindow(parent, title)
{
  TWindow MyChildWindow(this);

  TLayoutMetrics layoutMetrics;

  layoutMetrics.X.Absolute(lmLeft, 10);
  layoutMetrics.Y.Absolute(lmTop, 10);
  layoutMetrics.Width.PercentOf(lmParent, lmWidth, 60);
  layoutMetrics.Height.PercentOf(lmParent, lmHeight, 60);

  SetChildLayoutMetrics(MyChildWindow, layoutMetrics);
}
```

  Notice that the child window doesn't need any special functionality to be associated with a layout metrics object. The association is handled entirely by the layout window itself. The child window doesn't have to know anything about the relationship.

- *GetChildLayoutMetrics* returns **bool** and takes a *TWindow* **&** and a *TLayoutMetrics* **&** as parameters. This looks up the child window that is represented by the *TWindow* **&**. It then places the current layout metrics associated with that child window into the *TLayoutMetrics* object passed in. If *GetChildLayoutMetrics* doesn't find a child window that equals the window object passed in, it returns **false**.
- *RemoveChildLayoutMetrics* returns **bool** and takes a *TWindow* **&** for a parameter. This looks up the child window that is represented by the *TWindow* **&**. It then removes the child window and its associated layout metrics from the layout window's child list. If *RemoveChildLayoutMetrics* doesn't find a child window that equals the window object passed in, it returns **false**.

You must provide layout metrics for all child windows of a layout window. The layout window assumes that all of its children have an associated layout metrics object. Removing a child window from a layout window, or deleting the child window object automatically removes the associated layout metrics object.

## Frame windows

Frame windows (objects of class *TFrameWindow*) are specialized windows that support a *client window*. Frame windows are the basis for MDI and SDI frame windows, MDI child windows, and, along with *TLayoutWindow*, decorated frame windows.

Frame windows have an important role in ObjectWindows development: frame windows manage application-wide tasks like menus and tool bars. Client windows within the frame can be specialized to perform a single task. Changes you make to the frame window (for example, adding tool bars and status bars) don't affect the client windows.

## Constructing frame window objects

You can construct a frame window object using one of the two *TFrameWindow* constructors. These two constructors let you create new frame window objects along with new interface elements, and let you connect a new frame window object to an existing interface element.

### Constructing a new frame window

The first *TFrameWindow* constructor is used to create an entirely new frame window object:

```
TFrameWindow(TWindow *parent,
             const char far *title = 0,
             TWindow *clientWnd = 0,
             bool shrinkToClient = false,
             TModule *module = 0);
```

where:

- The first parameter is the window's parent window object. Use zero if the window you're creating is the main window (which doesn't have a parent window object). Otherwise, use a pointer to the parent window object. This is the only parameter that you *must* provide.
- The second parameter is the window title. This is the string that appears in the caption bar of the window. If you don't specify anything for the second parameter, no title is displayed in the title bar.
- The third parameter lets you specify a client window for the frame window. If you don't specify anything for the third parameter, by default the constructor gets a zero, meaning that there is no client window. Otherwise, pass a pointer to the client window object.
- The fourth parameter lets you specify whether the frame window should shrink to fit the client window. If you don't specify anything, by default the constructor gets **false**, meaning that it should not fit the frame to the client window.
- The fifth parameter is passed to the base class constructor as the *TModule* parameter for that constructor. This parameter defaults to 0.

Here are some examples of using this constructor:

```
void
TMyApplication::InitMainWindow()
{
  // default is for no client window
  SetMainWindow(new TFrameWindow(0, "Main Window"));
```

```
}

void
TMyApplication::InitMainWindow()
{
  // client window is TMyClientWindow
  SetMainWindow(new TFrameWindow(0, "Main window with client",
                                 new TMyClientWindow, true));
}
```

### Constructing a frame window alias

The second *TFrameWindow* constructor is used to connect an existing interface element to a new *TFrameWindow* object. This object is known as an *alias* for the existing window:

```
TFrameWindow(HWND hWnd, TModule *module);
```

where:

- The first parameter is the window handle of the existing interface element. This is the window the *TFrameWindow* object controls.
- The second parameter is passed to the base class constructor as the *TModule* parameter for that constructor. This parameter defaults to 0.

This is useful for creating window objects for existing windows. You can then manipulate any window as if it was an ObjectWindows-created window. This is useful in situations such as DLLs, when a non-ObjectWindows application calling into the DLL passes in an HWND. You can then construct a *TFrameWindow* alias for the HWND and proceed to call *TFrameWindow* member functions like normal.

The following example shows how to construct a *TFrameWindow* for an existing interface element and use that window as the main window:

```
void
TMyApplication::AddWindow(HWND hWnd)
{
  TFrameWindow* frame = new TFrameWindow(hWnd);
  TFrameWindow* tmp = SetMainWindow(frame);
  ShowWindow(GetMainWindow()->HWindow, SW_SHOW);
  tmp->ShutDownWindow();
}
```

When you use the second constructor for *TFrameWindow*, it sets the flag *wfAlias*. You can tell whether a window element was constructed from its window object or whether it's actually an alias by calling the function *IsFlagSet* with the *wfAlias* flag. For example, suppose you don't know whether the function *AddWindow* in the last example has executed yet. If your main window is *not* an alias, *AddWindow* hasn't executed. If your main window *is* an alias, *AddWindow* has executed:

```
void
TMyApplication::CheckAddExecute()
{
  if(GetMainWindow()->IsFlagSet(wfAlias))
    // MainWindow is an alias; AddWindow has executed
```

```
    else
      // MainWindow is not an alias; AddWindow has not executed
  }
```

See page 32 for more information on windows object attributes.

## Modifying frame windows

Many frame window attributes can be set after the object has been constructed. You can change and query object attributes using the functions discussed on page 32. You can also use the *TWindow* functions discussed on page 32. *TFrameWindow* provides an additional set of functions for modifying frame windows:

- *AssignMenu* is typically used to set up a window's menu before the interface element has been created, such as in the *InitMainWindow* function or the window object's constructor or *SetupWindow* function.
- *SetMenu* sets the window's menu handle to the HMENU parameter passed in.
- *SetMenuDescr* sets the window's menu description to the *TMenuDescr* parameter passed in.
- *GetMenuDescr* returns the current menu description.
- *MergeMenu* merges the current menu description with the *TMenuDescr* parameter passed in.
- *RestoreMenu* restores the window's menu from *Attr.Menu*.
- *SetIcon* sets the icon in the module passed as the first parameter to the icon passed as a resource in the second parameter.

For more specific information on these functions, refer to the *ObjectWindows Reference Guide*.

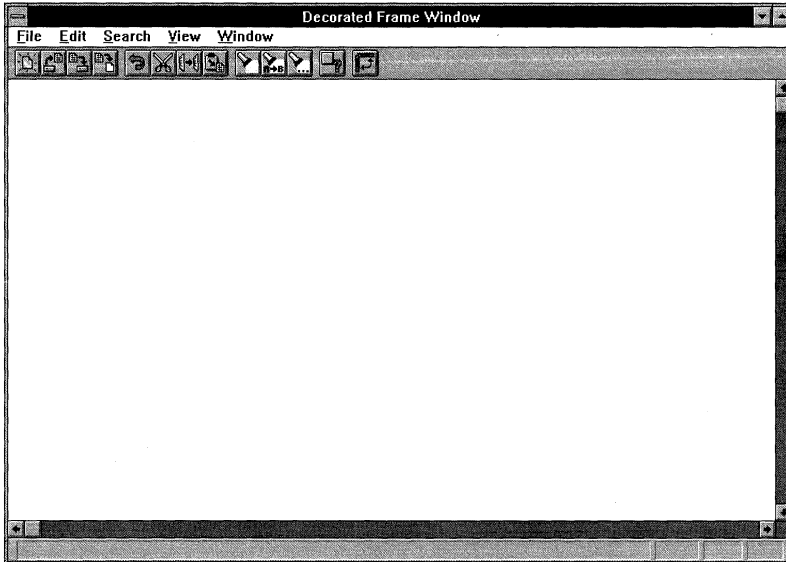
# Decorated frame windows

This section discusses decorated frame windows. Decorated frame windows are encapsulated in *TDecoratedFrame*, which is derived from *TFrameWindow* and *TLayoutWindow*. Decorated frame windows provide all the functionality of frame windows and layout, but in addition provide:

- Support for adding controls (known as *decorations*) to the frame of the window
- Automatic adjustment of child windows to accommodate the placement of decorations

Figure 7.2 shows a sample decorated frame window.

**Figure 7.2** Sample decorated frame window




## Constructing decorated frame window objects

*TDecoratedFrame* has only one constructor. Except for the fourth parameter, this constructor looks nearly identical to the first *TFrameWindow* constructor described on page 76.

```
TDecoratedFrame(TWindow* parent,
                const char far* title,
                TWindow* clientWnd,
                bool trackMenuSelection = false,
                TModule* module = 0);
```

where:

- The first parameter is the window's parent window object. Use zero if the window you're creating is the main window (which doesn't have a parent window object). Otherwise use a pointer to the parent window object. This is the only parameter that you *must* provide.
- The second parameter is the window title. This string appears in the caption bar of the window. If you don't specify anything for the second parameter, no title is displayed in the title bar.
- The third parameter lets you specify a pointer to a client window for the frame window. If you don't specify anything for the third parameter, by default the constructor gets a zero, meaning that there is no client window.
- The fourth parameter lets you specify whether menu commands should be tracked. When tracking is on, the window tries to pass a string to the window's status bar. The string passed has the same resource name as the currently selected menu choice. You

should not turn on menu selection tracking unless you have a status bar in your window. If you don't specify anything, by default the constructor gets **false**, meaning that it should not track menu commands.

- The fifth parameter is passed to the base class constructor as the *TModule* parameter for that constructor. This parameter defaults to 0.

## Adding decorations to decorated frame windows

You can use the methods for modifying windows described on pages 32 and 78 to modify the basic attributes of a decorated frame window. *TDecoratedFrame* provides the extra ability to add decorations using the *Insert* member function.

To use the *Insert* member function, you must first construct a control to be inserted. Valid controls include control bars (*TControlBar*), status bars (*TStatusBar*), button gadgets (*TButtonGadget*), and any other control type based on *TWindow*.

Once you have constructed the control, use the *Insert* function to insert the control into the decorated frame window. The *Insert* function takes two parameters: a reference to the control and a location specifier. *TDecoratedFrame* provides the **enum** *TLocation*. *TLocation* has four possible values: *Top*, *Bottom*, *Left*, and *Right*.

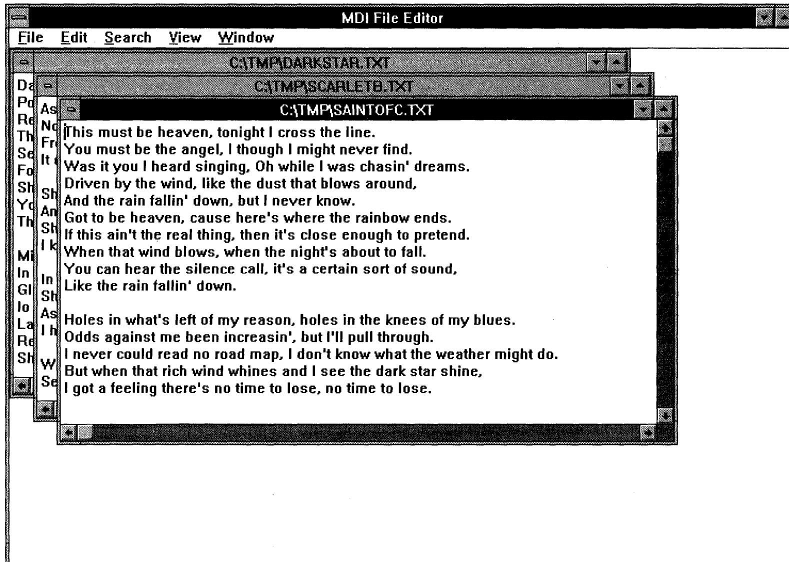
Suppose you want to construct a status bar to add to the bottom of your decorated frame window. The code would look something like this:

```
TStatusBar* sb = new TStatusBar(0, TGadget::Recessed,
                                TStatusBar::CapsLock |
                                TStatusBar::NumLock |
                                TStatusBar::Overtype);

TDecoratedFrame* frame = new TDecoratedFrame(0,
                                             "Decorated Frame",
                                             0,
                                             true);

frame->Insert(*sb, TDecoratedFrame::Bottom);
```

# MDI windows

Multiple-document interface, or MDI, windows are part of the MDI interface for managing multiple windows or views in a single frame window. MDI lets the user work with a number of child windows at the same time. Figure 7.3 shows a sample MDI application.

**Figure 7.3** Sample MDI application




# MDI applications

Certain components are present in every MDI application. Most evident is the main window, called the *MDI frame window*. Within the frame window's client area is the *MDI client window*, which holds child windows called *MDI child windows*. When using the Doc/View classes, the application can put views into MDI windows. See Chapter 10 for more information on the Doc/View classes.

## MDI Window menu

An MDI application usually has a menu item labeled Window that controls the MDI child windows. The Window menu usually has items like Tile, Cascade, Arrange, and Close All. The name of each open MDI child window is automatically added to the end of this menu, and the currently selected window is checked.

## MDI child windows

MDI child windows have some characteristics of an overlapped window. An MDI child window can be maximized to the full size of its MDI client window, or minimized to an icon that sits inside the client window. MDI child windows never appear outside their client or frame windows. Although MDI child windows can't have menus attached to them, they can have a *TMenuDescr* that the frame window uses as a menu when that child is active. The caption of each MDI child window is often the name of the file associated with that window; this behavior is optional and under your control.

## MDI in ObjectWindows

ObjectWindows defines classes for each type of MDI window:

- *TMDIFrame*
- *TMDIClient*
- *TMDIChild*

In ObjectWindows, the MDI frame window owns the MDI client window, and the MDI client window owns each of the MDI child windows.

*TMDIFrame*'s member functions manage the frame window and its menu. ObjectWindows first passes commands to the focus window and then to its parent, so the client window can process the frame window's menu commands. Because *TMDIFrame* doesn't have much specialized behavior, you'll rarely have to derive your own MDI frame window class; instead, just use an instance of *TMDIFrame*. Since *TMDIChild* is derived from *TFrameWindow*, it can be a frame window with a client window. Therefore, you can create specialized windows that serve as client windows in a *TMDIChild*, or you can create specialized *TMDIChild* windows. The preferred style is to use specialized clients with the standard *TMDIChild* class. The choice is yours, and depends on your particular application.

# Building MDI applications

Follow these steps to building an MDI application in ObjectWindows:

1. Create an MDI frame window
2. Add behavior to an MDI client window
3. Create MDI child windows

The ObjectWindows *TMDIXxx* classes handle the MDI-specific behavior for you, so you can concentrate on the application-specific behavior you want.

## Creating an MDI frame window

The MDI frame window is always an application's main window, so you construct it in the application object's *InitMainWindow* member function. MDI frame windows differ from other frame windows in the following ways:

- An MDI frame is always a main window, so it never has a parent. Therefore, *TMDIFrame*'s constructor doesn't take a pointer to a parent window object as a parameter.
- An MDI frame must have a menu, so *TMDIFrame*'s constructor takes a menu resource identifier as a parameter. With non-MDI main frame windows, you'd call *AssignMenu* to set the windows menu. *TMDIFrame*'s constructor makes the call for you. Part of what *TMDIFrame::AssignMenu* does is search the menu for the child-window menu, by searching for certain menu command IDs. If it finds a Window menu, new child window titles are automatically added to the bottom of the menu.

A typical *InitMainWindow* for an MDI application looks like this:

```
void
TMDIApp::InitMainWindow()
{
  SetMainWindow(new TMDIFrame("MDI App", ID_MENU, *new TMyMDIClient));
}
```

The example creates an MDI frame window titled "MDI App" with a menu from the ID_MENU resource. The ID_MENU menu should have a child-window menu. The MDI client window is created from the *TMyMDIClient* class.

## Adding behavior to an MDI client window

Since you usually use an instance of *TMDIFrame* as your MDI frame window, you need to add application-wide behavior to your MDI client window class. The frame window owns menus and tool bars but passes the commands they generate to the client window and to the application. A common message-response function would respond to the File | Open menu command to open another MDI child window.

### Manipulating child windows

*TMDIClient* has several member functions for manipulating MDI child windows. Commands from an MDI application's child-window menu control the child windows. *TMDIClient* automatically responds to those commands and performs the appropriate action:

**Table 7.4** Standard MDI child-window menu behavior

| Action | Menu command ID | *TMDIClient* member function |
|---|---|---|
| Cascade | CM_CASCADECHILDREN | *CmCascadeChildren* |
| Tile | CM_TILECHILDREN | *CmTileChildren* |
| Tile Horizontally | CM_TILECHILDRENHORIZ | *CmTileChildrenHoriz* |
| Arrange Icons | CM_ARRANGEICONS | *CmArrangeIcons* |
| Close All | CM_CLOSECHILDREN | *CmCloseChildren* |

The header file owl\mdi.h includes owl\mdi.rh for your applications. owl\mdi.rh is a resource header file that defines the menu command IDs listed in Table 7.4. When you design your menus in your resource script, be sure to include owl\mdi.rh to get those IDs.

MDI child windows shouldn't respond to any of the child-window menu commands. The MDI client window takes care of them.

## Creating MDI child windows

There are two ways to create MDI child windows: automatically in *TMDIClient::InitChild* or manually elsewhere.

### Automatic child window creation

*TMDIClient* defines the *CmCreateChild* message response function to respond to the CM_CREATECHILD message. *CmCreateChild* is commonly used to respond to an MDI application's File | New menu command. *CmCreateChild* calls *CreateChild*, which calls *InitChild* to construct an MDI child window object, and finally calls that object's *Create* member function to create the MDI child window interface element.

If your MDI application uses CM_CREATECHILD as the command ID to create new MDI child windows, then you should override *InitChild* in your MDI client window class to construct MDI child window objects whenever the user chooses that command:

```
TMDIChild*
TMyMDIClient::InitChild()
{
  return new TMDIChild(*this, "MDI child window");
}
```

Since *TMDIChild*'s constructor takes a reference to its parent window object, and not a pointer, you need to dereference the **this** pointer.

### Manual child window creation

You don't have to construct MDI child window objects in *InitChild*. If you construct them elsewhere, however, you must create their interface element yourself:

```
void
TMyMDIClient::CmFileOpen()
{
  new TMDIChild(*this, "")->Create();
}
```

Chapter

# 8

# Menu objects

ObjectWindows menu objects encapsulate menu resources and provide an interface for controlling and modifying the menu. Many applications use only a single menu assigned to the main window during its initialization. Other applications might require more complicated menu handling. ObjectWindows menu objects, encapsulated in the *TMenu*, *TSystemMenu*, *TPopupMenu*, and the *TMenuDescr* classes, give you an easy way to create and manipulate menus, from basic functionality to complex menu merging.

This chapter discusses the following tasks you can perform with menu objects:

- Constructing menu objects
- Modifying menu objects
- Querying menu objects
- Using system menu objects
- Using pop-up menu objects
- Using menu objects with frame windows

## Constructing menu objects

*TMenu* has several constructors to create menu objects from existing windows or from menu resources. After the menu is created, you can add, delete, or modify it using *TMenu* member functions. Table 8.1 lists the constructors you can use to create menu objects.

**Table 8.1** TMenu constructors for creating menu objects

| *TMenu* constructor | Description |
|---|---|
| TMenu() | Creates an empty menu. |
| TMenu(HWND) | Creates a menu object representing the window's current menu. |
| TMenu(HMENU) | Creates a menu object from an already-loaded menu. |
| TMenu(LPCVOID*) | Creates a menu object from a menu template in memory. |
| TMenu(HINSTANCE, TResID) | Creates a menu object from a resource. |

# Modifying menu objects

After you create a menu object, you can use *TMenu* member functions to modify it. Table 8.2 lists the member functions you can call to modify menu objects.

**Table 8.2** TMenu constructors for modifying menu objects

| *TMenu* member function | Description |
|---|---|
| **Adding menu items:** | |
| AppendMenu(uint, uint, const char*) | Adds a menu item to the end of the menu. |
| AppendMenu(uint, uint, const TBitmap&) | Adds a bitmap as a menu item at the end of the menu. |
| InsertMenu(uint, uint, uint, const char*) | Adds a menu item to the menu after the menu item of the given ID. |
| InsertMenu(uint, uint, uint, const TBitmap&) | Adds a bitmap as a menu item after the menu item of the given ID. |
| **Modifying menu items:** | |
| ModifyMenu(uint, uint, uint, const char*) | Changes the given menu item. |
| ModifyMenu(uint, uint, uint, const TBitmap&) | Changes the given menu item to a bitmap. |
| **Enabling and disabling menu items:** | |
| EnableMenuItem(uint, uint) | Enables or disables the given menu item. |
| **Deleting and removing menu items:** | |
| DeleteMenu(uint, uint) | Removes the menu item from the menu it is part of. Deletes it if it's a pop-up menu. |
| RemoveMenu(uint, uint) | Removes the menu item from the menu but not from memory. |
| **Checking menu items:** | |
| CheckMenuItem(uint, uint) | Check or unchecks the menu item. |
| SetMenuItemBitmaps(uint, uint, const TBitmap*, const TBitmap*) | Specifies the bitmap to be displayed when the given menu item is checked and unchecked. |
| **Displaying pop-up menus:** | |
| TrackPopupMenu(uint, int, int, int, HWND, TRect*)<br>TrackPopupMenu(uint, TPoint&, int, HWND, TRect*) | Displays the menu as a pop-up menu at the given location on the specified window. |

After modifying the menu object, you should call the window object's *DrawMenuBar* member function to update the menu bar with the changes you've made.

# Querying menu objects

*TMenu* has a number of member functions and member operators you can call to find out information about the menu object and its menu. You might need to call one of the query member functions before you call one of the modify member functions. For

example, you need to call *GetMenuCheckmarkDimensions* before calling *SetMenuItemBitmaps*.

Table 8.3 lists the menu-object query member functions.

**Table 8.3** TMenu constructors for querying menu objects

| *TMenu* member function | Description |
|---|---|
| **Querying the menu object as a whole:** | |
| **operator** *uint()* | Returns the menu's handle as a uint. |
| **operator** *HMENU()* | Returns the menu's handle as an HMENU. |
| IsOK() | Checks if the menu is OK (has a valid handle). |
| GetMenuItemCount() | Returns the number of items in the menu. |
| GetMenuCheckMarkDimensions(TSize&) | Gets the size of the bitmap used to display the check mark on checked menu items. |
| **Querying items in the menu:** | |
| GetMenuItemID(int) | Returns the ID of the menu item at the specified position. |
| GetMenuState(uint, uint) | Returns the state flags of the specified menu item. |
| GetMenuString(uint, char*, int, uint) | Gets the text of the given menu item. |
| GetSubMenu(int) | Returns the handle of the menu at the given position. |

# Using system menu objects

ObjectWindows' *TSystemMenu* class lets you modify a window's System menu. *TSystemMenu* is derived from *TMenu* and differs from it only in its constructor, which takes a window handle and a **bool** flag. If the flag is **true**, the current System menu is deleted and a menu object representing the unmodified menu that's put in its place is created. If the flag is **false**, the menu object represents the current System menu. By default this flag is **false**.

You can use all the member functions inherited from *TMenu* to manipulate the System menu. For example, the following example shows how to add an About menu choice to the System menu.

```
void
TSysMenuFrame::SetupWindow()
{
  TFrameWindow::SetupWindow();

  // Append about menu item to system menu.
  TSystemMenu sysMenu(HWindow);
  sysMenu.AppendMenu(MF_SEPARATOR, 0, (LPSTR)0);
  sysMenu.AppendMenu(MF_STRING, CM_ABOUT, "&About...");
}
```

Notice that the System menu is modified in the *SetupWindow* function of the window object. The System menu should be modified before the window is created. It's usually easiest to do this simply by overriding the base window class' *SetupWindow* function.

## Using pop-up menu objects

You can use *TPopupMenu* to create a pop-up menu that you can add to an existing menu structure or pop up anywhere in the window. Like *TSystemMenu*, *TPopupMenu* is derived from *TMenu* and differs from it only in its constructor, which creates an empty pop-up menu. You can then add whatever menu items you like using the *AppendMenu* function.

Once you've created a pop-up menu, you can use *TrackPopupMenu* to display it as a "free-floating" menu. *TrackPopupMenu* creates a pop-up menu at a particular location in your window. There are two forms of this function.

```
bool TrackPopupMenu(uint flags, int x, int y, int rsvd, HWND wnd, TRect* rect = 0);
bool TrackPopupMenu(uint flags, TPoint& point, int rsvd, HWND wnd, TRect* rect = 0);
```

where:

- *flags* specifies the relative location of the pop-up menu. It can be one of the following values:
  - TPM_CENTERALIGN
  - TMP_LEFTALIGN
  - TPM_RIGHTALIGN
  - TPM_LEFTBUTTON
  - TPM_RIGHTBUTTON
- *x* and *y* specify the screen location of the pop-up menu. In the second form of *TrackPopupMenu*, *point* does the same thing, combining *x* and *y* into a single *TPoint* object. The menu is then created relative to this point, depending on the value of *flags*.
- *rsvd* is a reserved value and must be set to 0.
- *wnd* is the handle to the window that receives messages about the menu.
- *rect* defines the area that the user can click without dismissing the menu.

The following example shows a window class that displays a pop-up menu in response to a right mouse button click.

```
class TPopupMenuFrame : public TFraemeWindow
{
  public:
    TPopupMenuFrame(TWindow* parent, const char *name);

  protected:
    TPopupMenu PopupMenu;
    void EvRButtonDown(uint modKeys, TPoint& point);

  DECLARE_RESPONSE_TABLE(TPopupMenuFrame);
};

DEFINE_RESPONSE_TABLE1(TSysMenuFrame, TFrameWindow)
  EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
```

```
TPopupMenuFrame::TPopupMenuFrame(TWindow* parent, const char *name)
  : TFrameWindow(parent, name)
{
  PopupMenu.AppendMenu(MF_STRING, CM_FILENEW, "Create new file");
  PopupMenu.AppendMenu(MF_STRING, CM_FILEOPEN, "Open file");
  PopupMenu.AppendMenu(MF_STRING, CM_FILESAVE, "Save file");
  PopupMenu.AppendMenu(MF_STRING, CM_FILESAVEAS, "Save file under new name");
  PopupMenu.AppendMenu(MF_STRING, CM_PENSIZE, "Change pen size");
  PopupMenu.AppendMenu(MF_STRING, CM_PENCOLOR, "Change pen color");
  PopupMenu.AppendMenu(MF_STRING, CM_ABOUT, "&About...");
  PopupMenu.AppendMenu(MF_STRING, CM_EXIT, "Exit Program");
}

void
TPopupMenuFrame::EvRButtonDown(uint /* modKeys */, TPoint& point)
{
  PopupMenu.TrackPopupMenu(TPM_LEFTBUTTON, point, 0, HWindow);
}
```

# Using menu objects with frame windows

ObjectWindows frame window objects (*TFrameWindow* and *TFrameWindow*-derived classes) provide a number of functions that you can use to assign, change, and modify menus. There are two ways to manipulate frame window menus:

- Directly assigning or changing the frame window's main menu. This is typically how you work with menus when you have a single menu that doesn't use menu merging.
- Assigning and merging the frame window's menu descriptor with that of client and child windows. Menu descriptors are objects that divide the menu bar into functional groups and permit easy merging and removal of pop-up menus.

These methods of using menu objects are described in the next sections.

## Adding menu resources to frame windows

It was fairly common practice in ObjectWindows 1.0 to assign a menu resource directly to the *Attr.Menu* member of a frame window; for example,

```
Attr.Menu = MENU_1;
```

ObjectWindows no longer permits this type of assignment; you should instead use the *AssignMenu* function. *AssignMenu* is defined in the *TFrameWindow* class, and is available in any class derived from *TFrameWindow*, such as *TMDIFrame*, *TMDIChild*, *TDecoratedFrame*, and *TFloatingFrame*.

The *AssignMenu* function takes a *TResId* for its only parameter and returns **true** if the assignment operation was successful. *AssignMenu* is declared **virtual**, so you can override it in your own *TFrameWindow*-derived classes. Here's what the previous example looks like when the *AssignMenu* function is used:

```
AssignMenu(MENU_1);
```

You can also change the menu after the frame window has been created. To change the frame window's menu, call the window object's *SetMenu* function.

```
SetMenu(MENU_2);
```

## Using menu descriptors

Managing menus—adding menus for child windows, merging menus, and so on—can be a tedious and confusing chore. ObjectWindows simplifies menu management with objects known as menu descriptors. Menu descriptors divide the menu bar into six groups, which correspond to conventional ways of arranging functions on a menu bar:

- File
- Edit
- Container
- Object
- Window
- Help

Organizing menus into functional groups makes it easy to insert a new menu into an existing menu bar. For example, consider an MDI application, such as Step 11 of the ObjectWindows tutorial in the *ObjectWindows Tutorial* manual. The frame and client windows provide menus that let the user perform general application functions such as opening files, managing windows, and so on. The child windows handle the menu commands for functions specific to a particular drawing, such as setting the line width and color.

In the tutorial, the menu stays the same, but menu items handled by the child windows are grayed out when no child window is available to handle the command. Another way to handle this would be to have the menu bar populated only with the menus handled by the frame and client windows. Then, when a child window is opened, the menus handled by the child window would be merged into the existing menu bar. The figures below show how this looks to the user. Figure 8.1 shows the application with no child windows open. Notice that there are only four pop-up menus on the menu bar.

**Figure 8.1** Menu descriptor application without child windows open

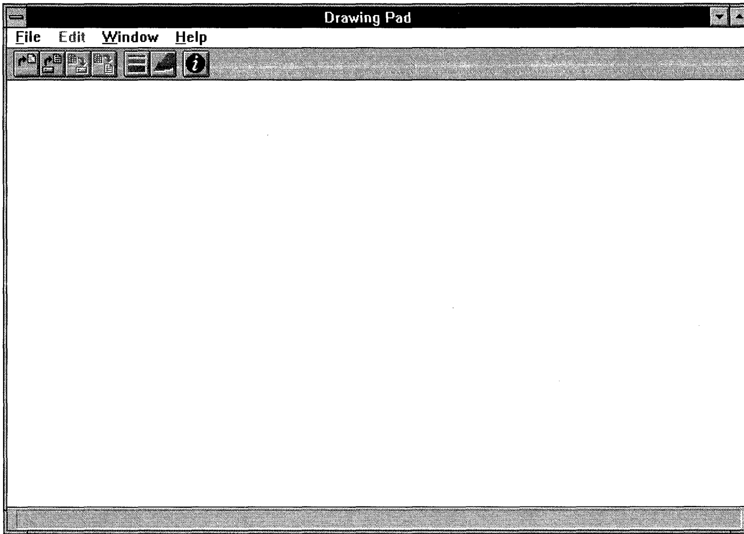
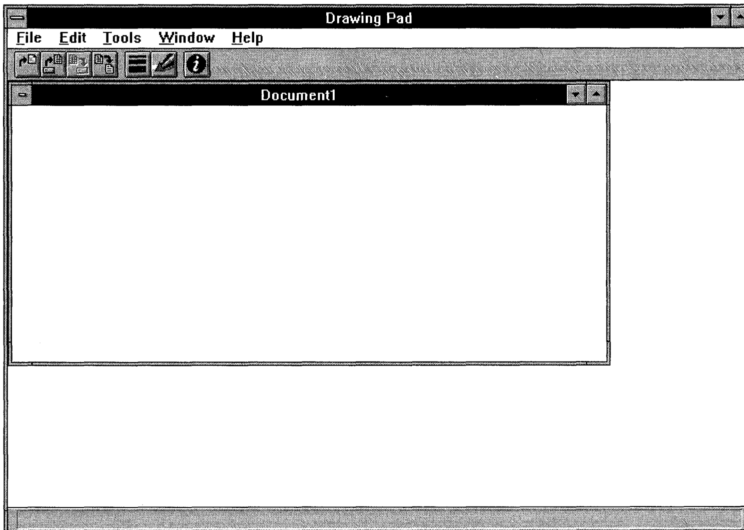



Figure 8.2 shows the application once one or more child windows have been opened. Notice the extra pop-up menu labeled Tools. The Tools menu is merged into the main menu bar only when there is a child window where the tools can be used.

**Figure 8.2** Menu descriptor application with child windows open




Adding menu descriptors to an application is a simple process.

- Set the menu descriptor for the frame window's menu bar by calling the frame window's *SetMenuDescr* function.

- When creating a new child window, set the child's menu descriptor by calling the child's *SetMenuDescr* function. Once the child window is created, ObjectWindows automatically merges the menu from the child with the frame window's menu bar while the child is active. Note that different MDI child windows in the same application can have different menu descriptors. This is useful when the child windows contain different kinds of documents.

Creating menu descriptors and using the menu descriptor handling functions is described in the next sections.

## Creating menu descriptors

The *TMenuDescr* class implements the ObjectWindows menu descriptor functionality. Menu descriptors take a menu resource and place the separate pop-up menus in the resource into six functional groups. The naming of the groups is arbitrary in that you are not restricted to putting only menus of a certain functional type into a particular group. However, the naming convention does reflect standard conventions of menu item placement. These names are contained in the *TGroup* **enum** defined in the *TMenuDescr* class:

- *FileGroup*
- *EditGroup*
- *ContainerGroup*
- *ObjectGroup*
- *WindowGroup*
- *HelpGroup*

These groups are arranged consecutively on the menu bar from left to right. When another menu descriptor is merged with the existing menu bar, the new pop-up menus are merged according to their groups. For example, consider the example show Figure 8.1 and Figure 8.2. The original three pop-up menus are placed in the following menu groups:

- The File menu is placed in the *FileGroup* group.
- The Window menu is placed in the *WindowGroup* group.
- The Help menu is placed in the *HelpGroup* group.

When the child window is created, its pop-up menu, called Tools, is placed in the *EditGroup* group. Then, when the menus are merged, the child window's menu is automatically placed between the File menu and the Window menu.

## Constructing menu descriptor objects

There are a number of different constructors for *TMenuDescr*. These are described in Table 8.4.

**Table 8.4** TMenuDescr constructors

| Constructor | Function |
|---|---|
| *TMenuDescr(TResId id, TModule* module = ::Module)* | Creates a menu descriptor from the menu resource identified by *id*. The grouping of the pop-up menus are determined by the occurrence of separators at the menu level (that is, separators inside of a pop-up menu are disregarded for grouping purposes) in the menu resource. This is discussed in more detail in the next section. |
| *TMenuDescr(TResId id, int fg, int eg, int cg, int og, int wg, int hg, TModule* module = ::Module);*<br><br>*TMenuDescr(HMENU hMenu, int fg, int eg, int cg, int og, int wg, int hg, TModule* module = ::Module);* | Creates a menu descriptor from the menu resource identified by *id* or *hMenu*. The separate pop-ups in the resource are then placed in groups according to the values of *fg*, *eg*, *cg*, *og*, *wg*, and *hg*. The total of all the values of *fg*, *eg*, *cg*, *og*, *wg*, and *hg* should be equivalent to the number of pop-ups in the menu resource. The *fg*, *eg*, *cg*, *og*, *wg*, and *hg* parameters correspond to the groups defined in the *TMenuDescr::TGroup* **enum**.<br>You can place more than one pop-up in a single group, and you don't have to place a pop-up in every group. For example, suppose you have a menu resource with a File menu, a Window menu, and a Help menu, all contained in the menu resource COMMANDS. You want to insert the File menu in the *FileGroup* group, the Window menu in the *WindowGroup* group, and the File menu in the *FileGroup* group. The constructor would look something like this:<br>`TMenuDescr md(COMMANDS, 1, 0, 0, 0, 1, 1);` |
| *TMenuDescr()* | Creates a default menu constructor without menu resources or any group counts. |
| *TMenuDescr(const TMenuDescr& original)* | Creates a copy of the menu descriptor object *original*. |

## Creating menu groups in menu resources

The *TMenuDescr* class provides two ways to set up the groups that your various pop-up menus belong in:

- Explicitly numbering the menu resources in the *TMenuDescr* constructor
- Placing separators at the pop-up menu level in the menu resource

Earlier versions of ObjectWindows provided only the first method. The second method is new in ObjectWindows 2.5 . This method is more flexible, eliminating the need to modify the *TMenuDescr* constructor whenever you add or remove a pop-up menu in your menu resource.

To set up groups in your menu resource, you need to put separators at the pop-up menu level. This means placing the separators *outside* of pop-up definitions. These separators have meaning only to the *TMenuDescr* constructor and don't cause any changes in the appearance of your menu bar. Separators inside pop-up menus are treated normally, that is, they appear in the pop-up menu as separator bars between menu choices.

The following example shows how a menu resource might be divided up into groups using separators in the menu resource. The menu resource is divided up into the requisite six groups, with four of the groups containing actual pop-up menus—the File

menu, the Edit menu, the Window menu, and the Help menu. The other two groups are empty.

```
IDM_COMMANDS MENU
{
  POPUP "File"
  {
    MENUITEM "&New\aCtrl+N", CM_FILENEW
    MENUITEM "&Open\aCtrl+O", CM_FILEOPEN
    MENUITEM "&Save\aCtrl+S", CM_FILESAVE
    MENUITEM "Save &as...", CM_FILESAVEAS
    MENUITEM SEPARATOR
    MENUITEM "&Print\aCtrl+P", CM_FILEPRINT
  }
  MENUITEM SEPARATOR
  POPUP "&Edit"
  {
    MENUITEM "&Undo\aCtrl+Z", CM_EDITUNDO
    MENUITEM Separator
    MENUITEM "&Cut\aCtrl+X", CM_EDITCUT
    MENUITEM "C&opy\aCtrl+C", CM_EDITCOPY
    MENUITEM "&Paste\aCtrl+V", CM_EDITPASTE
    MENUITEM "&Delete\aDel", CM_EDITDELETE
  }
  MENUITEM SEPARATOR
  MENUITEM SEPARATOR
  MENUITEM SEPARATOR
  POPUP "&Window"
  {
    MENUITEM "&Cascade", CM_CASCADECHILDREN
    MENUITEM "&Tile", CM_TILECHILDREN
    MENUITEM "Arrange &Icons", CM_ARRANGEICONS
    MENUITEM "C&lose All", CM_CLOSECHILDREN
    MENUITEM "Add &View", CM_VIEWCREATE
  }
  MENUITEM SEPARATOR
  POPUP "&Help"
  {
    MENUITEM "&About", CM_ABOUT
  }
}
```

## Merging menus with menu descriptors

To use menu descriptors for menu merging, you need to set your frame window's menu descriptor sometime before the creation of the window, usually during the *InitMainWindow* function. Then whenever you wish to merge a child window's menu or menus with that of its parent, you set the child window's menu descriptor before creating the child. When child is created, its menu descriptor is automatically merged with the parent.

You set a window's menu descriptor using the *SetMenuDescr* function. *SetMenuDescr* is inherited from *TFrameWindow*. It returns **void** and takes a **const** *TMenuDescr* reference

as its only parameter. The following example shows how you might create and set the menu descriptors for the examples shown in Figure 8.1 and Figure 8.2.

```
class TMenuDescrApp : public TApplication
{
  public:
    TMenuDescrApp(const char* name) : TApplication(name) {}

    void InitMainWindow()
    {
      SetMainWindow(Frame = new TMDIFrame(Name, COMMANDS, *new TMenuDescrMDIClient));
      Frame->SetMenuDescr(TMenuDescr(COMMANDS));
    }

  protected:
    TMDIFrame* Frame;
};

void
TMenuDescrMDIClient::CmAddMenu1()
{
  TMDIChild *child = new TMDIChild(*this, "Child Window 1", new TMenuDescrWindow, true);
  child->SetMenuDescr(TMenuDescr(IDM_MENU1));
  child->Create();
}
```

Chapter

# 9

# Dialog box objects

Dialog box objects are interface objects that encapsulate the behavior of dialog boxes. The *TDialog* class supports the initialization, creation, and execution of all types of dialog boxes. As with window objects derived from *TWindow*, you can derive specialized dialog box objects from *TDialog* for each dialog box your application uses.

ObjectWindows also supplies classes that encapsulate Windows' *common dialog boxes*. Windows provides common dialog boxes as a way to let users choose file names, fonts, colors, and so on.

This chapter covers the following topics:

- Using dialog box objects
- Using a dialog box as your main window
- Manipulating controls in dialog boxes
- Associating interface objects with controls
- Using common dialog boxes

## Using dialog box objects

Using dialog box objects is a lot like using window objects. For simple dialog boxes that appear for only a short period of time, you can control the dialog box in one member function of the parent window. The dialog box object can be constructed, executed, and destroyed in the member function.

Using a dialog box object requires the following steps:

- Constructing the object
- Executing the dialog box
- Closing the dialog box
- Destroying the object

## Constructing a dialog box object

Dialog boxes are designed and created using a dialog box resource. You can use Borland's Resource Workshop or any other resource editor to create dialog box resources and bind them to your application. The dialog box resource describes the appearance and location of controls, such as buttons, list boxes, group boxes, and so on. The dialog box resource isn't responsible for the behavior of the dialog box; that's the responsibility of the application.

Each dialog box resource has an identifier that enables a dialog box object to specify which dialog box resource it uses. The identifier can be either a string or an integer. You pass this identifier to the dialog box constructor to specify which resource the object should use.

### Calling the constructor

To construct a dialog box object, create it using a pointer to a parent window object and a resource identifier (the resource identifier can be either string or integer based) as the parameters to the constructor:

```
TDialog dialog1(this, "DIALOG_1");
  ⋮
TDialog dialog2(this, IDD_MY_DIALOG);
```

The parent window is almost always **this**, since you normally construct dialog box objects in a member function of a window object. If you don't construct a dialog box object in a window object, use the application's main window as its parent, because that is the only window object always present in an ObjectWindows application:

```
TDialog mySpecialDialog(GetApplication()->GetMainWindow(), IDD_DLG);
```

The exception to this is when you specify a dialog box object as a client window in a *TFrameWindow* or *TFrameWindow*-based constructor. The constructor passes the dialog box object to the *TFrameWindow::Init* function, which automatically sets the dialog box's parent. See page 102.

## Executing a dialog box

Executing a dialog box is analogous to creating and displaying a window. However, because dialog boxes are usually displayed for a shorter period of time, some of the steps can be abbreviated. This depends on whether the dialog box is a modal or modeless dialog box.

### Modal dialog boxes

Most dialog boxes are *modal*. While a modal dialog box is displayed, the user can't select or use its parent window. The user must use the dialog box and close it before proceeding. A modal dialog box, in effect, freezes the operation of the rest of the application.

Use *TDialog::Execute* to execute a dialog box modally. When the user closes the dialog box, *Execute* returns an integer value indicating how the user closed the dialog box. The return value is the identifier of the control the user pressed, such as IDOK for the OK

button or IDCANCEL for a Cancel button. If the dialog box object was dynamically allocated, be sure to delete the object.

The following example assumes you have a dialog resource IDD_MY_DIALOG, and that the dialog box has two buttons, an OK button that sends the identifier value IDOK and a Cancel button that sends some other value:

```
if (TMyDialog(this, IDD_MY_DIALOG).Execute() == IDOK)
  // User pressed OK
else
  // User pressed Cancel
```

Only the object is deleted when it goes out of scope, not the dialog box resource. You can create and delete any number of dialog boxes using only a single dialog box resource.

## Modeless dialog boxes

Unlike a modal dialog box, you can continue to use other windows in your application while a modeless dialog box is open. You can use a modeless dialog box to let the user continue to perform actions, find information, and so on, while still using the dialog box.

Use *TDialog::Create* to execute a dialog box modelessly. When using *Create* to execute a dialog box, you must explicitly make the dialog box visible by either specifying the WS_VISIBLE flag for the resource style or using the *ShowWindow* function to force the dialog box to display itself.

For example, suppose your resource script file looks something like this:

```
DIALOG_1 DIALOG 18, 18, 142, 44
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog 1"
{
  PUSHBUTTON "Button", IDOK, 58, 23, 25, 16
}
```

Now suppose that you try to create this dialog box modelessly using the following code:

```
  ⋮
TDialog dialog1(this, "DIALOG_1");
dialog1.Create();
  ⋮
```

This dialog box wouldn't appear on your screen. To make it appear, you'd have to do one of two things:

- Change the style of the dialog box to have the WS_VISIBLE flag set:

  ```
  STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
  ```

- Add the *ShowWindow* function after the call to *Create*:

  ```
    ⋮
  TDialog dialog1(this, "DIALOG_1");
  dialog1.Create();
  dialog1.ShowWindow(SW_SHOW);
    ⋮
  ```

The *TDialog::CmOk* and *TDialog::CmCancel* functions close the dialog box and delete the object. These functions handle the IDOK and IDCANCEL messages, usually sent by the OK and Cancel buttons, in the *TDialog* response table. The *CmOk* function calls *CloseWindow* to close down the modeless dialog box. The *CmCancel* function calls *Destroy* with the IDCANCEL parameter. Both of these functions close the dialog box. If you override either *CmOk* or *CmCancel*, you need to either call the base class *CmOk* or *CmCancel* function in your overriding function or perform the closing and cleanup operations yourself.

Alternately, you can create your dialog box object in the dialog box's parent's constructor. This way, you create the dialog box object just once. Furthermore, any changes made to the dialog box state, such as its location, active focus, and so on, are kept the next time you open the dialog box.

Like any other child window, the dialog box object is automatically deleted when its parent is destroyed. This way, if you close down the dialog box's parent, the dialog box object is automatically destroyed; you don't need to explicitly delete the object.

In the following code fragment, a parent window constructor constructs a dialog box object, and another function actually creates and displays the dialog box modelessly:

```
class TParentWindow : public TFrameWindow
{
  public:
    TParentWindow(TWindow* parent, const char* title);
    void CmDOIT();

  protected:
    TDialog *dialog;
};

  ⋮

void
TParentWindow::CmDO_IT()
{
  dialog = new TDialog(this, IDD_EMPLOYEE_INFO);
  dialog->Create();
}
```

## Using autocreation with dialog boxes

You can use autocreation to let ObjectWindows do the work of explicitly creating your child dialog objects for you. By creating the objects in the constructor of a *TWindow*-derived class and specifying the **this** pointer as the parent, the *TWindow*-derived class builds a list of child windows. This also happens when the dialog box object is a data member of the parent class. Then, when the *TWindow*-derived class is created, it attempts to create all the children in its list that have the *wfAutoCreate* flag turned on. This results in the children appearing onscreen at the same time as the parent window.

Turn on the *wfAutoCreate* flag using the function *EnableAutoCreate*. Turn off the *wfAutoCreate* flag using the function *DisableAutoCreate*.

*TWindow* uses *Create* for autocreating its children. Thus any dialog boxes created with autocreation are modeless dialog boxes.

Just as with regular modeless dialog boxes, if you're using autocreation to turn your dialog boxes on, you must make your dialog box visible. But with autocreation you must turn the WS_VISIBLE flag on in the resource file. You can't use the *ShowWindow* function to enable autocreation.

The following code shows how to enable autocreation for a dialog box:

```
class TMyFrame : public TFrameWindow
{
  public:
    TDialog *dialog;
    TMyFrame(TWindow *, const char far *);
};

TMyFrame::TMyFrame(TWindow *parent, const char far *title)
{
  Init(parent, true);
  dialog = new TDialog(this, "MYDIALOG");

  // For the next line to work properly, the WS_VISIBLE attribute
  // must be specified for the MYDIALOG resource.

  dialog->EnableAutoCreate();
}
```

When you execute this application, the dialog box is automatically created for you. See page 37 for more information on autocreation.

## Managing dialog boxes

Dialog boxes differ from other child windows, such as windows and controls, in that they are often displayed and destroyed many times during the life of their parent windows but are rarely displayed or destroyed at the same time as their parents. Usually, an application displays a dialog box in response to a menu selection, mouse click, error condition, or other event.

Therefore, you must be sure to not repeatedly construct new dialog box objects without deleting previous ones. Remember that when you construct a dialog box object in its parent window object's constructor or include the dialog box as a data member of the parent window object, the dialog box object is inserted into the child-window list of the parent and deleted when the parent is destroyed.

You can retrieve data from a dialog box at any time, as long as the dialog box object still exists. You'll do this most often in the dialog box object's *CmOK* member function, which is called when the user presses the dialog box's OK button.

## Handling errors executing dialog boxes

Like window objects, a dialog box object's *Create* and *Execute* member functions can throw the C++ exception *TXWindow*. This exception is usually thrown when the dialog

box can't be created, usually because the specified resource doesn't exist or because of insufficient memory.

You can rely on the global exception handler that ObjectWindows installs when your application starts to catch *TXWindow*, or you can install your own exception handler. To install your own exception handler, place a **try/catch** block around the code you want to protect. For example, if you want to know if your function *DoStuff* produces an error, the code would look something like this:

```
try
{
  DoStuff();
}

catch(TWindow::TXWindow& e)
{
  // You can do whatever exception handling you like here.
  MessageBox(0, e.why().c_str(),
             "Error", MB_OK);
}
```

ObjectWindows exception handling is explained in more detail in Chapter 6.

## Closing the dialog box

Every dialog box must have a way for the user to close it. For modal dialog boxes, this is usually an OK or Cancel button, or both. *TDialog* has the event response functions *CmOk* and *CmCancel* to respond to those buttons.

*CmOk* calls *CloseWindow*, which calls *CanClose* to see if it's OK to close the dialog box. If *CanClose* returns **true**, *CloseWindow* transfers the dialog's data and closes the dialog box by calling *CloseWindow*.

*CmCancel* calls *Destroy*, which closes the dialog box. No checking of *CanClose* is performed, and no transfer is done.

To verify the input in a dialog box, you can override the dialog box object's *CanClose* member function. Also see the description of the *TInputValidator* classes in Chapter 15. If you override *CanClose*, be sure to call the parent *TWindow::CanClose* function, which handles calling *CanClose* for child windows.

# Using a dialog box as your main window

To use a dialog box as your main window, it's best to make the main window a frame window that has your dialog box as a client window. To do this, derive an application class from *TApplication*. Aside from a constructor, the only function necessary for this purpose is *InitMainWindow*. In the *InitMainWindow* function, construct a frame window object, specifying a dialog box as the client window. In the five-parameter *TFrameWindow* constructor, pass a pointer to the client window as the third parameter. Your code should look something like this:

```
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\dialog.h>

class TMyApp : public TApplication
{
  public:
    TMyApp(char *title) : TApplication(title) {}
    void InitMainWindow();
};

void
TMyApp::InitMainWindow()
{
  SetMainWindow(new TFrameWindow(0, "My App",
                              new TDialog(0, "MYDIALOG"), true));
}

int
OwlMain(int argc, char* argv[])
{
  return TMyApp("My App").Run();
}
```

The *TFrameWindow* constructor turns autocreation on for the dialog box object that you pass as a client, regardless of the state you pass it in. For more information on autocreation for dialog boxes, see page 100.

You also must make sure the dialog box resource has certain attributes:

- Destroying your dialog object does not destroy the frame. You must destroy the frame explicitly.
- You can no longer dynamically add resources directly to the dialog, because it isn't the main window. You must add the resources to the frame window. For example, suppose you added an icon to your dialog using the *SetIcon* function. You now must use the *SetIcon* function for your frame window.
- You can't specify the caption for your dialog in the resource itself anymore. Instead, you must set the caption through the frame window.
- You must set the style of the dialog box as follows:
  - Visible (WS_VISIBLE)
  - Child window (WS_CHILD)
  - No Minimize and Maximize buttons, drag bars, system menus, or any of the other standard frame window attributes

# Manipulating controls in dialog boxes

Almost all dialog boxes have (as child windows) controls such as edit controls, list boxes, buttons, and so on. Those controls are created from the dialog box's resource.

There is a two-way communication between a dialog box object and its controls. In one direction, the dialog box needs to manipulate its controls; for example, to fill a list box. In the other direction, it needs to process and respond to the messages the controls generate; for example, when the user selects an item from a list box. To learn about responding to controls, see Chapter 3.

Chapter 11 describes using controls in more detail, and also discusses how to use controls in windows instead of dialog boxes.

### Communicating with controls

Windows defines a set of control messages that are sent from the application back to Windows. For example, list-box messages include LB_GETTEXT, LB_GETCURSEL, and LB_ADDSTRING. Control messages specify the specific control and pass along information in *wParam* and *lParam* arguments. Each control in a dialog resource has an identifier, which you use to specify the control to receive the message. To send a control message, you can call *SendDlgItemMessage*. For example, the following member function adds the specified string to the list box using the LB_ADDSTRING message:

```
void
TTestDialog::FillListBox(const char far* string)
{
  SendDlgItemMessage(ID_LISTBOX, LB_ADDSTRING, 0, (LPARAM)string);
}
```

It's rarely necessary to communicate with controls like this; ObjectWindows control classes provide member functions to perform the same actions. This section discusses the mechanisms used to perform this communication only to enhance your understanding of the process. Although *TListBox::AddString* does basically the same thing as this function and is easier to understand, this shows how you can use *SendDlgItemMessage* to force actions.

## Associating interface objects with controls

Because a dialog box is created from its resource, you don't use C++ code to specify what it looks like or the controls in it. Although this lets you create the dialog box visually, it makes it harder to manipulate the controls from your application. ObjectWindows lets you "connect" or *associate* controls in a dialog box with interface objects. Associating controls with control objects lets you do two things:

- Provide specialized responses to messages. For example, you might want an edit control that allows only digits to be entered, or you might want a button that changes styles when it's pressed.
- Use member functions and data members to manipulate the control. This is easier and more object-oriented than using control messages.

## Control objects

To associate a control object with a control element, you can define a pointer to a control object as a data member and construct a control object in the dialog box object's constructor. Control classes such as *TButton* have a constructor that takes a pointer to the parent window object and the control's resource identifier. In the following example, *TTestDialog*'s constructor creates a *TButton* object from the resource ID_BUTTON:

```
TTestDialog::TTestDialog(TWindow* parent, const char* resID)
  : TDialog(parent, resID), TWindow(parent)
{
  new TButton(this, ID_BUTTON);
}
```

You can also define your own control class, derived from an existing control class (if you want to provide specialized behavior). In the following example, *TBeepButton* is a specialized *TButton* that overrides the default response to the BN_CLICKED notification code. A *TBeepButton* object is associated with the ID_BUTTON button resource.

```
class TBeepButton : public TButton
{
  public:
    TBeepButton(TWindow* parent, int resId) : TButton(parent, resId) {}

    void BNClicked();  // BN_CLICKED

  DECLARE_RESPONSE_TABLE(TBeepButton);
};

DEFINE_RESPONSE_TABLE1(TBeepButton, TButton)
  EV_NOTIFY_AT_CHILD(BN_CLICKED, BNClicked),
END_RESPONSE_TABLE;

void
TBeepButton::BNClicked()
{
  MessageBeep(-1);
}
  ⋮
TBeepDialog::TBeepDialog(TWindow* parent, const char* name)
  : TDialog(parent, name), TWindow(parent)
{
  button = new TBeepButton(this, ID_BUTTON);
}
```

Unlike setting up a window object, which requires two steps (construction and creation), associating an interface object with an interface element requires only the construction step. This is because the interface element already exists: it's loaded from the dialog box resource. You just have to tell the constructor which control from the resource to use, using its resource identifier.

## Setting up controls

You can't manipulate controls by, for example, adding strings to a list box or setting the font of an edit control until the dialog box object's *SetupWindow* member function executes. Until *TDialog::SetupWindow* has called *TWindow::SetupWindow*, the dialog box's controls haven't been associated with the corresponding objects. Once they're associated, the objects' *HWindow* data members are valid for the controls.

In this example, the *AddString* function isn't called until the base class *SetupWindow* function is called:

```
class TDerivedDialog : public TDialog
{
  public:
    TDerivedDialog(TWindow* parent, TResId resId)
      : TDialog(parent, resId), TWindow(parent)
    {
      listbox = new TListBox(this, IDD_LISTBOX);
    }

  protected:
    TListBox* listbox;
};

void
TDerivedDialog::SetupWindow()
{
  TDialog::SetupWindow();
  listbox->AddString("First entry");
}
```

# Using dialog boxes

A Windows application often needs to prompt the user for file names, colors, or fonts. ObjectWindows provides classes that make it easy to use dialog boxes, including Windows' common dialog boxes. The following table lists the different types of dialog boxes and the ObjectWindows class that encapsulates each one.

**Table 9.1** ObjectWindows-encapsulated dialog boxes

| Type | ObjectWindows class |
|---|---|
| Color | *TChooseColorDialog* |
| Font | *TChooseFontDialog* |
| File open | *TFileOpenDialog* |
| File save | *TFileSaveDialog* |
| Find string | *TFindDialog* |
| Input from user | *TInputDialog* |
| Printer abort dialog | *TPrinterAbortDlg* |

**Table 9.1** ObjectWindows-encapsulated dialog boxes (continued)

| Type | ObjectWindows class |
|---|---|
| Printer control | *TPrintDialog* |
| Replace string | *TReplaceDialog* |

# Using input dialog boxes

Input dialog boxes are simple dialog boxes that prompt the user for a single line of text input. You can run input dialog boxes as either modal or modeless dialog boxes, but you'll usually run them modally. Input dialog box objects have a dialog box resource associated with them, provided in the resource script file owl\inputdia.rc. Your application's .RC file must include owl\inputdia.rc.

When you construct an input dialog box object, you specify a pointer to the parent window object, caption, prompt, and the text buffer and its size. The contents of the text buffer is the default input text. When the user chooses OK or presses ***Enter***, the line of text entered is automatically transferred into the character array. Here's an example:

```
char patientName[33] = "";

TInputDialog(this, "Patient name",
             "Enter the patient's name:",
             patientName, sizeof(patientName)).Execute();
```

In this example, *patientName* is a text buffer that gets filled with the user's input when the user chooses OK. It's initialized to an empty string for the default text.

# Using common dialog boxes

The common dialog boxes encapsulate the functionality of the Windows common dialog boxes. These dialog boxes let the user choose colors, fonts, file names, find and replace strings, print options, and more. You construct, execute, and destroy them similarly. The material in this section describes the common tasks; the material in the following sections describes the tasks specific to each type of common dialog box.

## Constructing common dialog boxes

Each common dialog box class has a nested class called *TData*. *TData* contains some common housekeeping members and data specific to each type of common dialog box. For example, *TChooseColorDialog::TData* has members for the color being chosen and an array for a set of custom colors. The following table lists the two members common to all *TData* nested classes.

**Table 9.2** Common dialog box TData members

| Name | Type | Description |
|---|---|---|
| Flags | *uint32* | A set of common dialog box-specific flags that control the appearance and behavior of the dialog box. For example, CC_SHOWHELP is a flag that tells the color selection common dialog box to display a Help button the user can press to get context-sensitive Help. Full information about the various flags is available in the *ObjectWindows Reference Guide*. |
| Error | *uint32* | This is an error code if an error occurred while processing a common dialog box; it's zero if no error occurred. *Execute* returns IDCANCEL both when the user chose Cancel and when an error occurred, so you should check *Error* to determine whether an error actually occurred. |

Each common dialog box class has a constructor that takes a pointer to a parent window object, a reference to that class' *TData* nested class, and optional parameters for a custom dialog box template, title string, and module pointer.

Here's a sample fragment that constructs a common color selection dialog box:

```
TChooseColorDialog::TData colors;
static TColor custColors[16] =
{
  0x010101L, 0x101010L, 0x202020L, 0x303030L,
  0x404040L, 0x505050L, 0x606060L, 0x707070L,
  0x808080L, 0x909090L, 0xA0A0A0L, 0xB0B0B0L,
  0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L
};

colors.CustColors = custColors;
colors.Flags = CC_RGBINIT;
colors.Color = TColor::Black;
if (TChooseColorDialog(this, colors).Execute() == IDOK)
  SetColor(colors.Color);
```

Once the user has chosen a new color in the dialog box and pressed OK, that color is placed in the *Color* member of the *TData* object.

## Executing common dialog boxes

Once you've constructed the common dialog box object, you should execute it (for a modal dialog box) or create it (for a modeless dialog box). The following table lists whether each type of common dialog box must be modal or modeless.

**Table 9.3** Common dialog box TData members

| Type | Modal or modeless | Run by calling |
|---|---|---|
| Color | Modal | *Execute* |
| Font | Modal | *Execute* |
| File open | Modal | *Execute* |
| File save | Modal | *Execute* |
| Find | Modeless | *Create* |

**Table 9.3** Common dialog box TData members (continued)

| Type | Modal or modeless | Run by calling |
|---|---|---|
| Find/replace | Modeless | *Create* |
| Printer | Modal | *Execute* |

You must check *Execute*'s return value to see whether the user chose OK or Cancel, or to determine if an error occurred:

```
TChooseColorDialog::TData colors;
TChooseColorDialog colorDlg(this, colors);

if (colorDlg.Execute() == IDOK)
  // OK: data.Color == the color the user chose
  ⋮ // Some code here.
else if (data.Error)
  // error occurred
  ⋮ // Some code here.

MessageBox("Error in color dialog box!", GetApplication()->Name,
           MB_OK | MB_ICONSTOP);
```

## Using color common dialog boxes

The color common dialog box lets you choose and create colors for use in your application. For example, a paint application might use the color common dialog box to choose the color of a paint bucket.

*TChooseColorDialog::TData* has several members you must initialize before constructing the dialog box object:

**Table 9.4** Color common dialog box TData data members

| *TData* member | Type | Description |
|---|---|---|
| Color | *TColor* | The selected color. When you execute the dialog box, this specifies the default color. When the user closes the dialog box, this specifies the color the user chose. |
| CustColors | *TColor** | A pointer to an array of sixteen custom colors. On input, it specifies the default custom colors. On output, it specifies the custom colors the user chose. |

In the following example, a color common dialog box is used to set the window object's *Color* member, which is used elsewhere to paint the window. Note the use of the *TWindow::Invalidate* member function to force the window to be repainted in the new color.

```
void
TCommDlgWnd::CmColor()
{
  // use static to keep custom colors around between
  // executions of the color common dialog box
  static TColor custColors[16];
```

```
  TChooseColorDialog::TData choose;

  choose.Flags = CC_RGBINIT;
  choose.Color = Color;
  choose.CustColors = custColors;

  if(TChooseColorDialog(this, choose).Execute() == IDOK)
    Color = choose.Color;
  Invalidate();
}
```

For details about *TData::Flags* in the *TChooseColorDialog* class, see the *ObjectWindows Reference Guide*.

## Using font common dialog boxes

The font common dialog box lets you choose a font to use in your application, including its typeface, size, style, and so on. For example, a word processor might use the font common dialog box to choose the font for a paragraph.

*TChooseFontDialog::TData* has several members you must initialize before constructing the dialog box object:

**Table 9.5** Font common dialog box TData data members

| *TData* member | Type | Description |
|---|---|---|
| DC | *HDC* | A handle to the device context of the printer whose fonts you want to select, if you specify CF_PRINTERFONTS in *Flags*. Otherwise ignored. |
| LogFont | *LOGFONT* | A handle to a *LOGFONT* that specifies the font's appearance. When you execute the dialog box and specify the flag CF_INITTOLOGFONTSTRUCT, the dialog box appears with the specified font (or the closest possible match) as the default. When the user closes the dialog box, *LogFont* is filled with the selections the user made. |
| PointSize | **int** | The point size of the selected font (in tenths of a point). On input, it sets the size of the default font. On output, it returns the size the user selected. |
| Color | *TColor* | The color of the selected font, if the CF_EFFECTS flag is set. On input, it sets the color of the default font. On output, it holds the color the user selected. |
| Style | **char far*** | Lets you specify the style of the dialog. |
| FontType | *uint16* | A set of flags describing the styles of the selected font. Set only on output. |
| SizeMin | **int** | Specifies the minimum and maximum |
| SizeMax | **int** | Point sizes (in tenths of a point) the user can select, if the CF_LIMITSIZE flag is set. |

In this example, a font common dialog box is used to set the window object's *Font* member, which is used elsewhere to paint text in the window. Note how a new font object is constructed, using *TFont*.

```
void
TCommDlgWnd::CmFont()
```

```
{
  TChooseFontDialog::TData FontData;

  FontData.DC = 0;
  FontData.Flags = CF_EFFECTS | CF_FORCEFONTEXIST | CF_SCREENFONTS;
  FontData.Color = Color;
  FontData.Style = 0;
  FontData.FontType = SCREEN_FONTTYPE;

  FontData.SizeMin = 0;
  FontData.SizeMax = 0;

  if (TChooseFontDialog(this, FontData).Execute() == IDOK) {
    delete Font;
    Color = FontData.Color;
    Font = new TFont(&FontData.LogFont);
  }
  Invalidate();
}
```

## Using file open common dialog boxes

The file open common dialog box serves as a consistent replacement for the many different types of dialog boxes applications have used to open files.

*TOpenSaveDialog::TData* has several members you must initialize before constructing the dialog box object. You can either initialize them by assigning values, or you can use *TOpenSaveDialog::TData*'s constructor, which takes *Flags*, *Filter*, *CustomFilter*, *InitialDir*, and *DefExt* (the most common) as parameters with default arguments of zero.

**Table 9.6** File open and save common dialog box TData data members

| *TData* member | Type | Description |
|---|---|---|
| FileName | **char*** | The selected file name. On input, it specifies the default file name. On output, it contains the selected file name. |
| Filter | **char*** | The file name filters and filter patterns. Each filter and filter pattern is in the form: `filter|filter pattern|...` where *filter* is a text string that describes the filter and *filter pattern* is a DOS wildcard file name. You can repeat *filter* and *filter pattern* for as many filters as you need. You must separate them with `|` characters. |
| CustomFilter | **char*** | Lets you specify custom filters. |
| FilterIndex | **int** | Specifies which of the filters specified in *Filter* should be displayed by default. |
| InitialDir | **char*** | The directory to be displayed on opening the file dialog box. Use zero for the current directory. |
| DefExt | **char*** | Default extension appended to *FileName* if the user doesn't type an extension. If *DefExt* is zero, no extension is appended. |

In this example, a file open common dialog box prompts the user for a file name. If an error occurred (*Execute* returns IDCANCEL and *Error* returns nonzero), a message box is displayed.

```
void
TCommDlgWnd::CmFileOpen()
{
  TFileOpenDialog::TData FilenameData
    (OFN_FILEMUSTEXIST | OFN_HIDEREADONLY | OFN_PATHMUSTEXIST,
    "All Files (*.*)|*.*|Text Files (*.txt)|*.txt|",
    0, "", "*");

  if (TFileOpenDialog(this, FilenameData).Execute() != IDOK) {
    if (FilenameData.Errval) {
      char msg[50];
      wsprintf(msg, "GetOpenFileName returned Error #%ld", Errval);
      MessageBox(msg, "WARNING", MB_OK | MB_ICONSTOP);
    }
  }
}
```

## Using file save common dialog boxes

The file save common dialog box serves as a single, consistent replacement for the many different types of dialog boxes that applications have previously used to let users choose file names.

*TOpenSaveDialog::TData* is used by both file open and file save common dialog boxes.

In the following example, a file save common dialog box prompts the user for a file name to save under. The default directory is WINDOWS and the default extension is .BMP.

```
void
TCanvasWindow::CmFileSaveAs()
{
  TOpenSaveDialog::TData data
    (OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    "Bitmap Files (*.BMP)|*.bmp|",
    0,
    "\windows",
    "BMP");

  if (TFileSaveDialog(this, data).Execute() == IDOK) {
    // save data to file
    ifstream is(FileData->FileName);

    if (!is)
      MessageBox("Unable to open file", "File Error", MB_OK | MB_ICONEXCLAMATION);
     else
       // Do file output
  }
}
```

## Using find and replace common dialog boxes

The find and replace common dialog boxes let you search and optionally replace text in your application's data. These dialog boxes are flexible enough to be used for documents or even databases. The simplest way to use the find and replace common dialog boxes is to use the *TEditSearch* or *TEditFile* edit control classes; they implement an edit control that you can search and replace text in. If your application is text-based, you can also use the find and replace common dialog boxes manually.

### Constructing and creating find and replace common dialog boxes

Since the find and replace dialog boxes are modeless, you normally keep a pointer to them as a data member in your parent window object. This makes it easy to communicate with them.

The find and replace common dialog boxes are modeless. You should construct and create them in response to a command (for example, a menu item Search | Find or Search | Replace). This displays the dialog box and lets the user enter the search information.

*TFindReplaceDialog::TData* has the standard *Flags* members, plus members for holding the find and replace strings. See the *ObjectWindows Reference Guide* for more details about *Flags*.

The following example shows the pointer to the find dialog box in the parent window object and shows the command event response function that constructs and creates the dialog box.

```
class TDatabaseWindow : public TFrameWindow
{
    ⋮
    TFindReplaceDialog::TData SearchData;
    TFindReplaceDialog* SearchDialog;
    ⋮
};

void
TDatabaseWindow::CmEditFind()
{
  // If the find dialog box isn't already
  // constructed, construct and create it now
  if (!SearchDialog) {
    SearchData.Flags |= FR_DOWN; // default to searching down
    SearchDialog = new TFindDialog(this, SearchData)
    SearchDialog->Create();
  }
}
```

### Processing find-and-replace messages

Since the find and replace common dialog boxes are modeless, they communicate with their parent window object by using a registered message *FINDMSGSTRING*. You must write an event response function that responds to *FINDMSGSTRING*. That event

response function takes two parameters—a *WPARAM* and an *LPARAM*—and returns an *LRESULT*. The *LPARAM* parameter contains a pointer that you must pass to the dialog box object's *UpdateData* member function.

After calling *UpdateData*, you must check for the FR_DIALOGTERM flag. The common dialog box code sets that flag when the user closes the modeless dialog box. Your event response function should then zero the dialog box object pointer because it's no longer valid. You must construct and create the dialog box object again.

As long as the FR_DIALOGTERM flag wasn't set, you can process the *FINDMSGSTRING* message by performing the actual search. This can be as simple as an edit control object's *Search* member function or as complicated as triggering a search of a Paradox or dBASE table.

In this example, *EvFindMsg* is an event response function for a registered message. *EvFindMsg* calls *UpdateData* and then checks the FR_DIALOGTERM flag. If it wasn't set, *EvFindMsg* calls another member function to perform the search.

```
DEFINE_RESPONSE_TABLE1(TDatabaseWindow, TFrameWindow)
  ⋮
  EV_REGISTERED(FINDMSGSTRING, EvFindMsg),
END_RESPONSE_TABLE;
⋮
LRESULT TDatabaseWindow::EvFindMsg(WPARAM, LPARAM lParam)
{
  if (SearchDialog) {
    SearchDialog->UpdateData(lParam);
    // is the dialog box closing?
    if (SearchData.Flags & FR_DIALOGTERM) {
      SearchDialog = 0;
      SearchCmd = 0;
    } else
      DoSearch();
  }
  return 0;
}
```

## Handling a Find Next command

The find and replace common dialog boxes have a Find Next button that users can use while the dialog boxes are visible. Most applications also support a Find Next command from the Search menu, so users can find the next occurrence in one step instead of having to open the find dialog box and click the Find Next button. *TFindDialog* and *TReplaceDialog* make it easy for you to offer the same functionality.

Setting the FR_FINDNEXT flag has the same effect as clicking the Find Next button:

```
void
TDatabaseWindow::CmEditFindNext()
{
  SearchDialog->UpdateData();
  SearchData.Flags |= FR_FINDNEXT;
  DoSearch();
}
```

## Using printer common dialog boxes

There are two printer common dialog boxes. The *print job* dialog box lets you choose what to print, where to print it, the print quality, the number of copies, and so on. The *print setup* dialog box lets you choose among the installed printers on the system, the page orientation, and paper size and source.

*TPrintDialog::TData*'s members let you control the appearance and behavior of the printer common dialog boxes:

**Table 9.7** Printer common dialog box TData data members

| *TData* member | Type | Description |
|---|---|---|
| FromPage | **int** | The first page of output, if the PD_PAGENUMS flag is specified. On input, it specifies the default first page. On output, it specifies the first page the user chose. |
| ToPage | **int** | The last page of output, if the PD_PAGENUMS flag is specified. On input, it specifies the default last page number. On output, it specifies the last page number the user chose. |
| MinPage | **int** | The fewest number of pages the user can choose. |
| MaxPage | **int** | The largest number of pages the user can choose. |
| Copies | **int** | The number of copies to print. On input, the default number of copies. On output, the number of copies the user actually chose. |

In the following example, *CmFilePrint* executes a standard print job common dialog box and uses the information in *TPrintDialog::TData* to determine what to print. *CmFilePrintSetup* adds a flag to bring up the print setup dialog box automatically.

```
void
TCanvas::CmFilePrint()
{
  if (TPrintDialog(this, data).Execute() == IDOK)
    // Use TPrinter and TPrintout to print the drawing
}

void
TCanvas::CmFilePrintSetup()
{
  static TPrintDialog::TData data;
  data.Flags |= PD_PRINTSETUP;

  if (TPrintDialog(this, data, 0).Execute() == IDOK)
    // Print
}
```

Chapter

# 10

# Doc/View objects

ObjectWindows provides a flexible and powerful way to contain and manipulate data: the Doc/View model. The Doc/View model consists of three parts:

- Document objects, which can contain many different types of data and provide methods to access that data.
- View objects, which form an interface between a document object and the user interface and control how the data is displayed and how the user can interact with the data.
- An application-wide document manager that maintains and coordinates document objects and the corresponding view objects.

## How documents and views work together

This section describes the basic concept of the Doc/View model. If you're already familiar with these concepts or if you want more technical information, refer to the programming sections beginning on page 121.

The Doc/View model frees the programmer and the user from worrying about what type of data a file contains and how that data is presented on the screen. Doc/View associates data file types with a document class and a view class. The document manager keeps a list of associations between document classes and view classes. Each association is called a *document template* (note that document templates are *not* related to C++ templates).

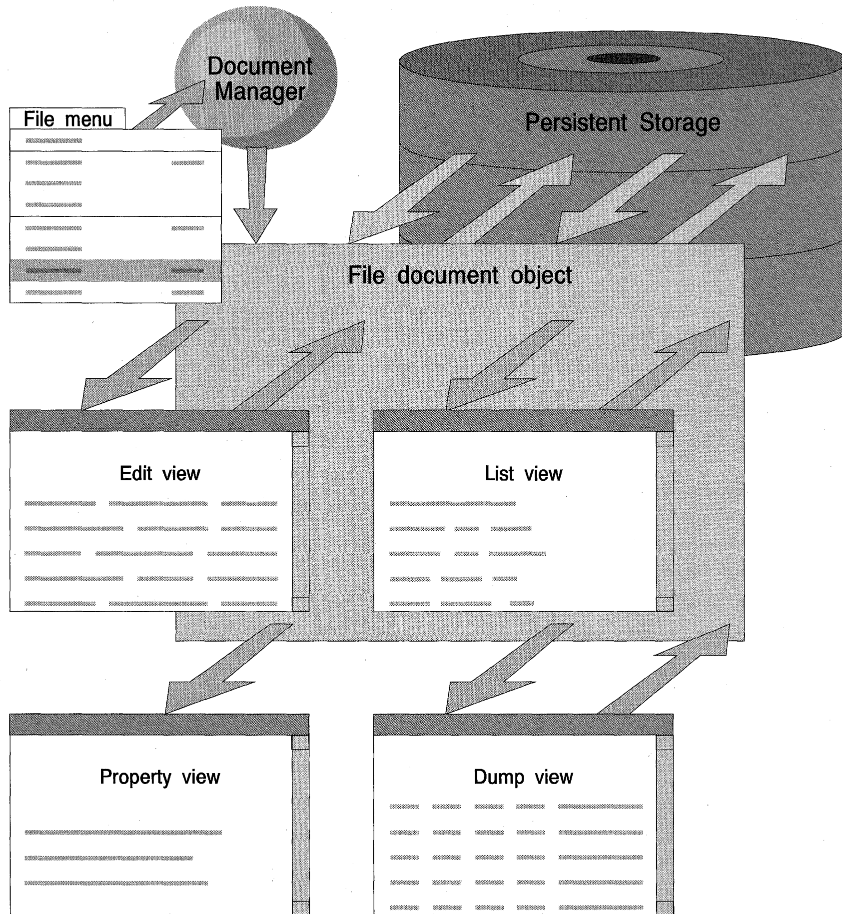
A document class handles data storage and manipulation. It contains the information that is displayed on the screen. A document object controls changes to the data and when and how the data is transferred to persistent storage (such as the hard drive, RAM disk, and so on).

When the user opens a document, whether by creating a new document or opening an existing document, the document is displayed using an associated view class. The view

class manages how the data is displayed and how the user interacts with the data onscreen. In effect, the view forms an interface between the display window and the document. Some document types might have only one associated view class; others might have several. Each different view type can be used to let the user interact with the data in a different way.

Table 10.1 illustrates the interaction between the document manager, a document class, and the document's associated views:

**Figure 10.1** Doc/View model diagram




This figure shows a file document object from the *TFileDocument* class, along with some associated views. The *TFileDocument* class is shown in the DOCVIEWX example. This example is in the directory *BC45*\EXAMPLES\OWL\OWLAPI\DOCVIEW, where *BC45* is the directory in which you installed Borland C++ 4.5.

## Documents

The traditional concept of a document and the Doc/View concept of a document differ in several important ways. The traditional concept of a document is generally like that of a word-processing file. It consists of text mixed with the occasional graphic, along with embedded commands to assist the word-processing program in formatting the document.

A Doc/View document differs quite significantly from the traditional concept of a document:

- The first distinction is between the contents of the two types of documents. Whereas the traditional document is mostly text with a few other bits of data, a Doc/View document can contain literally any type of data, such as text, graphics, sounds, multimedia files, and even other documents.
- The next distinction is in terms of presentation. Whereas the format of the traditional document is usually designed with the document's presentation in mind, a Doc/View document is completely independent of how it is displayed.
- The last distinction is that a document from a particular word-processing program is generally dependent on the format demanded by that program; documents are usually portable between different word-processing programs only after a tedious porting process. The intention of Doc/View documents is to let data be easily ported between different applications, even applications whose basic functions are highly divergent.

The basic functionality for a document object is provided in the ObjectWindows class *TDocument*. A more in-depth discussion of *TDocument* and how to use it as a basis for your own document classes is presented later in this chapter on page 128.

## Views

View objects enable document objects to present themselves to the world. Without a view object, you can't see or manipulate the document. But when you pair a document with a view object into a document template, you've got a functional piece of data and code that provides a graphic representation of the data stored in the document and a way to interact with and change that data.

The separation between the document and view also permits flexibility in when and how the data in document is modified. Although the data is manipulated through the view, the view only relays those changes on to the document. It is then up to the document to determine whether to change the data in the document (known as *committing* the changes) or discarding the changes (known as *reverting* back to the document).

Another advantage of using view objects instead of some sort of fixed-display method (such as a word-processing program) is that view objects offer the programmer and the user a number of different ways to display and manipulate the same document. Although you might need to provide only one view for a document type, you might also want to provide three or four views.

For example, suppose you create a document class to store graphic information, such as a picture or drawing. For a basic product, you might want to provide only one type of view, such as a view that draws the picture in a window and then lets the user "paint" and modify the picture. For a more advanced version, you might want to provide extra views; for example, the drawing could be displayed as a color separation, as a hexadecimal file, or even as a series of equations if the drawing was mathematically generated. To access these other views, users choose the type of view desired when they open the document. In all these scenarios, the document itself never changes.

The basic functionality for a view is provided in the ObjectWindows class *TView*. A more in-depth discussion of *TView* and how to use it as a basis for your own view classes is presented on page 134.

## Associating document and view classes

A document class is associated with its view class (or classes) by a document template. Document templates are created in two steps:

1. Define a template class by associating a document class with a view class.
2. Instantiate a template from a defined class.

The difference between these two steps is important. After you've defined a template class, you can create any number of instances of that template class. Each template associates *only* a document class and a view class. Each instance has a name, a default file extension, directory, flags, and file filters. Thus you could provide a single template class that associates a document with a view. You could then provide a number of different *instances* of that template class, where each instance handles files in a different default directory, with different extensions, and so on, still using the same document and view classes.

## Managing Doc/View

The document manager maintains the list of template instances used in your application and the list of current documents. Every application that uses Doc/View documents must have a document manager, but each application can have only one document manager at a time.

The document manager brings the Doc/View model together: document classes, view classes, and templates. The document manager provides a default File menu and default handling for each of the choices on the File menu:

**Table 10.1** Document manager's File menu

| Menu choice | Handling |
|---|---|
| New | Creates a new document. |
| Open... | Opens an existing document. |
| Save | Saves the current document. |
| As... | Saves the current document with a new name. |
| Revert To Saved | Reverts changes to the last document saved. |

**Table 10.1** Document manager's File menu (continued)

| Menu choice | Handling |
|---|---|
| Close | Closes the current document. |
| Exit | Quits the application, prompts to save documents. |

Once you've written your document and view classes, defined any necessary templates, and made instances of the required templates, all you still need to do is to create your document manager. When the document manager is created, it sets up its list of template instances and (if specified in the constructor) sets up its menu. Then whenever it receives one of the events that it handles, it performs the command specified for that event. The example on page 193 shows how to set up document manager for an application.

# Document templates

Document templates join together document classes and view classes by creating a new class. The document manager maintains a list of document templates that it uses when creating a new Doc/View instance. This section explains how to create and use document templates, including

- Designing document template classes
- Creating document registration tables
- Creating instances of document template classes
- Modifying existing document template classes

## Designing document template classes

You create a document template class using the DEFINE_DOC_TEMPLATE_CLASS macro. This macro takes three arguments:

- Document class
- View class
- Template class name

The document class should be the document class you want to use for data containment. The view class should be the view class you want to use to display the data contained in the document class. The template class name should be indicative of the function of the template. It cannot be a C++ keyword (such as **int**, **switch**, and so on) or the name of any other type in the application.

For example, suppose you've two document classes—one called *TPlotDocument*, which contains graphics data, and another called *TDataDocument*, which contains numerical data. Now suppose you have four view classes, two for each document class. For *TPlotDocument*, you have *TPlotView*, which displays the data in a *TPlotDocument* object as a drawing, and *THexView*, which displays the data in a *TPlotDocument* object as arrays of hexadecimal numbers. For *TDataDocument*, you have *TSpreadView*, which displays the data in a *TDataDocument* object much like a spreadsheet, and *TCalcView*,

which displays the data in a *TDataDocument* object after performing a series of calculations on the data.

To associate the document classes with their views, you would use the DEFINE_DOC_TEMPLATE_CLASS macro. The code would look something like this:

```
DEFINE_DOC_TEMPLATE_CLASS(TPlotDocument, TPlotView, TPlotTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TPlotDocument, THexView, THexTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TDataDocument, TSpreadView, TSpreadTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TDataDocument, TCalcView, TCalcTemplate);
```

As you can see from the first line, the existing document class *TPlotDocument* and the existing view class *TPlotView* are brought together and associated in a new class called *TPlotTemplate*. The same thing happens in all the other lines, so that you have four new classes, *TPlotTemplate*, *THexTemplate*, *TSpreadTemplate*, and *TCalcTemplate*. The next section describes how to use these new classes you've created.

## Creating document registration tables

Once you've defined a template class, you can create any number of instances of that class. You can use template class instances to provide different descriptions of a template, search for different default file names, look in different default directories, and so on. Each of these attributes of a template class instance is affected by the document registration table passed to the template class constructor.

Document registration tables let you specify the various attributes and place them in a single object. The object type is *TRegList*, although in normal cirsumstances, you shouldn't ever have to access this object directly. To create a registration table,

1. You always start a registration table definition with the BEGIN_REGISTRATION macro. This macro takes a single parameter, the name of the registration object. This name can be whatever you want it to be, although it should be somewhat descriptive of the particular template instance you want to create with it.
2. Once you've started the table you need to register a number of data items in the table. You can place these items in the table using the REGDATA macro. REGDATA takes two parameters. The first is a key that identifies the type of data, while the second is a string containing the actual data. The key should be a string composed of alphanumeric characters; you don't need to place quotes around this value. The actual data string can be any legal string; you *do* need to place quotes around this value. Also, you don't need to use commas or semicolons after the macros. There are three data items you need to enter in the table for an instance of a document template:
    1. The *description* value should be a short text description of the template class. It should be indicative of the type of data handled by the document class and how that data is displayed by the view class.
    2. The *extension* value should indicate the default file extension for documents of this type.
    3. The *docfilter* value should indicate the file name masks that should be applied to documents when searching through file names.

3 You also need to register a number of flags describing how this document type is to be opened or created. These document flags can be registered with the REGDOCFLAGS macro. REGDOCFLAGS takes a single parameter, the flags themselves. The flags specified can be one or more of the following:

**Table 10.2** Document creation mode flags

| Flag | Function |
|---|---|
| dtAutoDelete | Close and delete the document object when the last view is closed. |
| dtNoAutoView | Do not automatically create a default view. |
| dtSingleView | Allow only one view per document. |
| dtAutoOpen | Open a document upon creation. |
| dtHidden | Hide template from list of user selections. |

4 Once you've registered the necessary data items and the document mode flags, you can end the table definition with the END_REGISTRATION macro. This macro takes no parameters. You don't need to append a semicolon at the end of the line either.

The code below shows a sample registration table declaration. The resulting registration table is called *ListReg*, applies to a document template class described as a Line List, which has the default extension PTS, the default file-name mask **.pts*, is set to be automatically deleted when the last view on the document is closed, and is hidden from the list of documents available to the user.

```
BEGIN_REGISTRATION(ListReg)
  REGDATA(description,"Line List")
  REGDATA(extension,  ".PTS")
  REGDATA(docfilter,  "*.pts")
  REGDOCFLAGS(dtAutoDelete | dtHidden)
END_REGISTRATION
```

## Creating template class instances

Once you've created a document template class and a registration table, you're ready to create an actual instance of the template class. An instance of a document template class serves as an entry in the document manager's list of possible document and view combination that can be opened. Once this is in place, you can open documents of the type defined in the document template class and display the document in the specified view.

The signature of a template class constructor is always the same:

```
TplName name(TRegList& regTable);
```

where:

- *TplName* is the name you gave the template class when defining it.
- *name* is the name you want to give this instance (this name isn't very useful until you want to revise an existing template class instance).
- *regTable* is a registration table created using the BEGIN/END_REGISTRATION macros.

For example, suppose you've got the following template class definition:

```
DEFINE_DOC_TEMPLATE_CLASS(TPlotDocument, TPlotView, TPlotTemplate);
```

Now suppose you want to create three instances of this template class:

- One instance should have the description "Approved plots", for document files with the extension .PLT. You want to allow only a single view of the document and to automatically delete the document when the view is closed.
- Another instance should have the description "In progress", for document files with the extension .PLT. You want to automatically delete the document when the last view is closed.
- Another instance should have the description "Proposals", for document files with the extensions .PLT or .TMP (but with the default extension of .PLT). You want to keep this template hidden until the user has entered a password, and delete the document object when the last view is closed.

The code for creating these instances would look something like this:

```
BEGIN_REGISTRATION(aReg)
  REGDATA(description, "Approved plots",
  REGDATA(docfilter, "*.PLT",
  REGDATA(extension, "PLT",
  REGDOCFLAGS(dtSingleView | dtAutoDelete)
END_REGISTRATION

TPlotTemplate atpl(aReg);

BEGIN_REGISTRATION(bReg)
  REGDATA(description, "In progress",
  REGDATA(docfilter,"*.PLT",
  REGDATA(extension, "PLT",
  REGDOCFLAGS(dtAutoDelete);
END_REGISTRATION

TPlotTemplate btpl(bReg);

BEGIN_REGISTRATION(cReg)
  REGDATA(description, "Proposals",
  REGDATA(docfilter, "*.PLT; *.TMP",
  REGDATA(extension, "PLT",
  REGDOCFLAGS(dtHidden | dtAutoDelete);
END_REGISTRATION

TPlotTemplate *ctpl = new TPlotTemplate(cReg);
```

Just as in any other class, you can create both static and dynamic instances of a document template.

## Modifying existing templates

Once you've created an instance of a template class, you usually don't need to modify the template object. However, you might occasionally want to modify the properties with which you constructed the template. You can do this using these access functions:

- Use the *GetFileFilter* and *SetFileFilter* functions to get and set the string used to filter file names in the current directory.
- Use the *GetDescription* and *SetDescription* functions to get and set the text description of the template class.
- Use the *GetDirectory* and *SetDirectory* functions to get and set the default directory.
- Use the *GetDefaultExt* and *SetDefaultExt* functions to get and set the default file extension.
- Use the *GetFlags*, *IsFlagSet*, *SetFlag*, and *ClearFlag* functions to get and set the flag settings.

# Using the document manager

The document manager, an instance of *TDocManager* or a *TDocManager*-derived class, performs a number of tasks:

- Manages the list of current documents and registered templates
- Handles the standard File menu command events CM_FILENEW, CM_FILEOPEN, CM_FILESAVE, CM_FILESAVEAS, CM_FILECLOSE, and optionally CM_FILEREVERT
- Provides the file selection interface

To support the Doc/View model, a document manager must be attached to the application. This is done by creating an instance of *TDocManager* and making it the document manager for your application. The following code shows an example of how to attach a document manager to your application:

```
class TMyApp : public TApplication
{
  public:
    TMyApp() : TApplication() {}

    void InitMainWindow() {
       ⋮
      SetDocManager(new TDocManager(dmMDI | dmMenu));
       ⋮
    }
};
```

You can set the document manager to a new object using the *SetDocManager* function. *SetDocManager* takes a *TDocManager* **&** and returns **void**.

The document manager's public data and functions can be accessed through the document's *GetDocManager* function. *GetDocManager* takes no parameters and returns a *TDocManager* &. The document manager provides the following functions for creating documents and views:

- *CreateAnyDoc* presents all the visible templates, whereas the *TDocTemplate* member function *CreateDoc* presents only its own template.
- *CreateAnyView* filters the template list for those views that support the current document and presents a list of the view names, whereas the *TDocTemplate* member function *CreateView* directly constructs the view specified by the document template class.

Specialized document managers can be used to support other needs. For example, an OLE 2.0 server needs to support class factories that create documents and views through interfaces that are not their own. If the server is invoked with the embedded command-line flags, it doesn't bring up its own user interface and can attach a document manager that replaces the interface with the appropriate OLE support.

## Constructing the document manager

The constructor for *TDocManager* takes a single parameter that's used to set the mode of the document manager. You can open the document manager in one of two modes:

- In single-document interface (SDI) mode, you can have only a single document open at any time. If you open a new document while another document is already open, the document manager attempts to close the first document and replace it with the new document.
- In multiple-document interface (MDI) mode, you can have a number of documents and views open at the same time. Each view is contained in its own client window. Furthermore, each document can be a single document type presented by the same view class, a single document presented with different views, or even entirely different document types.

To open the document manager in SDI mode, call the constructor with the *dmSDI* parameter. To open the document manager in MDI mode, call the constructor with the *dmMDI* parameter.

There are three other parameters you can also specify:

- *dmMenu* specifies that the document manager should install its own File menu, which provides the standard document manager File menu and its corresponding commands.
- *dmSaveEnabled* enables the Save command on the File menu even if the document has not been modified.
- *dmNoRevert* disables the Revert command on the File menu.

Once you've constructed the document manager you cannot change the mode. The following example shows how to open the document manager in either SDI or MDI mode. It uses command-line arguments to let the user specify whether the document manager should open in SDI or MDI mode.

```
class TMyApp : public TApplication
{
  public:
    TMyApp() : TApplication() {}
    void InitMainWindow();
    int DocMode;
};

void
TMyApp::InitMainWindow()
{
  switch ((_argc > 1 && _argv[1][0]=='-' ? _argv[1][1] : (char)0) | ('S'^'s'))
  {
    case 's': DocMode = dmSDI; break; // command line: -s
    case 'm': DocMode = dmMDI; break; // command line: -m
    default : DocMode = dmMDI; break; // no command line
  }

  SetDocManager(new TDocManager(DocMode | dmMenu));
};
```

Thus, if the user starts the application with the **-s** option, the document manager opens in SDI mode. If the user starts the application with the **-m** option or with no option at all, the document manager opens in MDI mode.

## TDocManager event handling

If you specify the *dmMenu* parameter when you construct your *TDocManager* object, the document manager handles certain events on behalf of the documents. It does this by using a response table to process standard menu commands. These menu commands are provided by the document manager even when no documents are opened and regardless of whether you explicitly add the resources to your application. The File menu is also provided by the document manager.

The events that the document manager handles are

- CM_FILECLOSE
- CM_FILENEW
- CM_FILEOPEN
- CM_FILEREVERT
- CM_FILESAVE
- CM_FILESAVEAS
- CM_VIEWCREATE

In some instances, you might want to handle these events yourself. Because the document manager's event table is the last to be searched, you can handle these events at the view, frame, or application level. Another option is to construct the document manager without the dmMenu parameter. You must then provide functions to handle these events, generally through the application object or your interface object.

You can still call the document manager's functions through the *DocManager* member of the application object. For example, suppose you want to perform some action before

opening a file. Providing the function through your window class *TMyWindow* might look something like this:

```
class TMyApp : public TApplication
{
  public:
    TMyApp() : TApplication() {}
    void InitMainWindow();
    int DocMode;
};

void
TMyApp::InitMainWindow()
{
  // Don't specify dmMenu when constructing TDocManager
  SetDocManager(new TDocManager(dmMDI));
};

class TMyWindow : public TDecoratedMDIFrame
{
  public:
    TMyWindow();
    void CmFileOpen();

    // You also need to provide the other event handlers provided by the document manager.
     ⋮

  DECLARE_RESPONSE_TABLE(TMyWindow);
};

DEFINE_RESPONSE_TABLE1(TMyWindow, TDecoratedMDIFrame)
  EV_COMMAND(CM_FILEOPEN, CmFileOpen),
   ⋮
END_RESPONSE_TABLE;

void
TMyWindow::CmFileOpen()
{
  // Do your extra work here.
  GetApplication()->GetDocManager()->CmFileOpen();
}
```

# Creating a document class

The primary function of a document class is to provide callbacks for requested data changes in a view, to handle user actions as relayed through associated views, and to tell associated views when data has been updated. *TDocument* provides the framework for this functionality. The programmer needs only to add the parts needed for a specific application of the document model.

## Constructing TDocument

*TDocument* is an abstract base class that cannot be directly instantiated. Therefore you implement document classes by deriving them from *TDocument*.

You must call *TDocument*'s constructor when constructing a *TDocument*-derived class. The *TDocument* constructor takes only one parameter, a *TDocument* * that points to the parent document of the new document. If the document has no parent, you can either pass a 0 or pass no parameters; the default value for this parameter is 0.

## Adding functionality to documents

As a standard procedure, you should avoid overriding *TDocument* functions that aren't declared **virtual**. The document manager addresses all *TDocument*-derived objects as if they were actually *TDocument* objects. If you override a nonvirtual function, it isn't called when the document manager calls that function. Instead, the document manager calls the *TDocument* version of the function. But if you override a virtual function, the document manager correctly calls your class' version of the function.

The following functions are declared **virtual** in *TDocument*:

| | |
|---|---|
| *~TDocument* | *InStream* |
| *OutStream* | *Open* |
| *Close* | *Commit* |
| *Revert* | *RootDocument* |
| *SetDocPath* | *SetTitle* |
| *GetProperty* | *IsDirty* |
| *IsOpen* | *CanClose* |
| *AttachStream* | *DetachStream* |

You can override these functions to provide your own custom interpretation of the function. But when you do override a **virtual** function, you should be sure to find out what the base class function does. Where the base class performs some sort of essential function, you should call the base class version of the function from your own function; the base class versions of many functions perform a check of the document's hierarchy, including checking or notifying any child documents, all views, any open streams, and so on.

## Data access functions

*TDocument* provides a number of functions for data access. You can access data as a simple serial stream or in whatever way you design into your derived classes. The following sections describe the helper functions you can use to control when the document attempts data access operations.

## Stream access

*TDocument* provides two functions, *InStream* and *OutStream*, that return pointers to a *TInStream* and a *TOutStream*, respectively. The *TDocument* versions of these function both return a 0, because the functions actually perform no actions. To provide stream access for your document class you must override these functions, construct the appropriate stream class, and return a pointer to the stream object.

*TInStream* and *TOutStream* are abstract stream classes, derived from *TStream* and *istream* or *ostream*, respectively. *TStream* provides a minimal functionality to connect the stream to a document. *istream* and *ostream* are standard C++ iostreams. You must derive document-specific stream classes from *TInStream* and *TOutStream*. The *TInStream* and *TOutStream* classes are documented in the *ObjectWindows Reference Guide*. Here, though, is a simple description of the *InStream* and *OutStream* member functions. Both *InStream* and *OutStream* take two parameters in their constructors:

```
XXXStream(int mode, LPCSTR strmId = 0);
```

where *XXX* is either *In* or *Out*, *mode* is a stream opening mode identical to the *open_mode* flags used for *istream* and *ostream*, and *strmId* is a pointer to an existing stream object. Passing a valid pointer to an existing stream object in *strmId* causes that stream to be used as the document's stream object. Otherwise, the object opens a new stream object.

There are also two stream-access functions called *AttachStream* and *DetachStream*. Both of these functions take a reference to an existing (that is, already constructed and open) *TStream*-derived object. *AttachStream* adds the *TStream*-derived object to the document's list of stream objects, making it available for access. *DetachStream* searches the document's list of stream objects and deletes the *TStream*-derived object passed to it. Both of these functions have protected access and thus can be called only from inside the document object.

## Stream list

Each document maintains a list of open streams that is updated as streams are added and deleted. This list is headed by the *TDocument* data *StreamList*. *StreamList* is a *TStream* * that points to the first stream in the list. If there are no streams in the list, *StreamList* is 0. Each *TStream* object in the list has a member named *NextStream*, which points to the next stream in the stream list.

When a new stream is opened in a document object or an existing stream is attached to the object, it is added to the document's stream list. When an existing stream is closed in a document object or detached from the object, it is removed from the document's stream list.

## Complex data access

Streams can provide only simple serial access to data. In cases where a document contains multimedia files, database tables, or other complex data, you probably want more sophisticated access methods. For this purpose, *TDocument* uses two more access functions, *Open* and *Close*, which you can override to define your own opening and closing behavior.

The *TDocument* version of *Open* performs no actions; it always returns **true**. You can write your own version of *Open* to work however you want. There are no restrictions

placed on how you define opening a document. You can make it as simple as you like or as complex as necessary. *Open* lets you open a document and keep it open, instead of opening the document only on demand from one of the document's stream objects.

The *TDocument* version of *Close* provides a little more functionality than does *Open*. It checks any existing children of your document and tries to close them before closing your document. If you provide your own *Close*, the first thing you should do in that function is call the *TDocument* version of *Close* to ensure that all children have been closed before you close the parent document. Other than this one restriction, you are free to define the implementation of the *Close* function. Just as with *Open*, *Close* lets you close a document when you want it closed, as opposed to permitting the document's stream objects to close the document.

### Data access helper functions

*TDocument* also provides a number of functions that you can use to help protect your data:

*IsDirty* first checks to see whether the document itself is "dirty" (that is, modified but not updated) by checking the state of the data member *DirtyFlag*. It then checks whether any child documents are dirty, then whether any views are dirty. *IsDirty* returns **true** if any children or views are dirty.

*IsOpen* checks to see whether the document is held open or has any streams in its stream list. If the document is not open, *IsOpen* returns **false**. Otherwise, *IsOpen* returns **true**.

*Commit* commits any changes to your data to storage. Once you've called *Commit*, you cannot back out of any changes made. The *TDocument* version of this function checks any child documents and commits them to their changes. If any child document returns **false**, the *Commit* is aborted and returns **false**. All child documents must return **true** before the *Commit* function commits its own data. After all child documents have returned **true**, *Commit* flushes all the views for operations that might have taken place since the document last checked the views. Data in the document is updated according to the changes in the views and then saved. *Commit* then returns **true**.

*Revert* performs the opposite function from *Commit*. Instead of updating changes and saving the data, *Revert* clears any changes that have been made since the last time the data was committed. *Revert* also polls any child documents and aborts if any of the children return **false**. If all operations are successful, *Revert* returns **true**.

## Closing a document

Like most other objects, *TDocument* provides functions that let you safely close and destroy the object.

*~TDocument* does a lot of cleanup. First it destroys its children and closes all open streams and other resources. Then, in order, it detaches its attached template, closes all associated views, deletes its stream list, and removes itself from its parent's list of children if the document has a parent or, if it doesn't have a parent, removes itself from the document manager's document list.

In addition to a destructor, *TDocument* also provides a *CanClose* function to make sure that it's OK to close. *CanClose* first checks whether all its children can close. If any child returns **false**, *CanClose* returns **false** and aborts. If all child documents return **true**, *CanClose* calls the document manager function *FlushDoc*, which checks to see if the document is dirty. If the document is clean, *FlushDoc* and *CanClose* return **true**. If the document is dirty, *FlushDoc* opens a message box that prompts the user to either save the data, discard any changes, or cancel the close operation.

## Expanding document functionality

The functions described in this section include most of what you need to know to make a functioning document class. It is up to you to expand the functionality of your document class. Your class needs special functions for manipulating data, understanding and acting on the information obtained from the user through the document's associated view, and so on. All this functionality goes into your *TDocument*-derived class.

Because the Doc/View model is so flexible, there are no requirements or rules as to how you should approach this task. A document can handle almost any type of data because the Doc/View data-handling mechanism is a primitive framework, intended to be extended by derived classes. The base classes provided in ObjectWindows provide the functionality to support your extensions to the Doc/View model.

## Working with the document manager

*TDocument* provides two functions for accessing the document manager, *GetDocManager* and *SetDocManager*. *GetDocManager* returns a pointer to the current document manager. You can then use this pointer to access the data and function members of the document manager. *SetDocManager* lets you assign the document to a different document manager. All other document manager functionality is contained in the document manager itself.

## Working with views

*TDocument* provides two functions for working with views, *NotifyViews* and *QueryViews*. Both functions take three parameters, an **int** corresponding to an event, a **long** item, and a *TView* *. The meaning of the **long** item is dependent on the event and is essentially a parameter to the event. The *TView* * lets you exclude a view from your query or notification by passing a pointer to that view to the function. These two functions are your primary means of communicating information between your document and its views.

Both functions call views through the views' response tables. The general-purpose macro used for ObjectWindows notification events is EV_OWLNOTIFY. The response functions for EV_OWLNOTIFY events have the following signature:

```
bool FnName(long);
```

The **long** item used in the *NotifyViews* or *QueryViews* function call is used for the **long** parameter for the response function.

You can use *NotifyViews* to notify your child documents, their associated views, and the associated views of your root document of a change in data, an update, or any other event that might need to be reflected onscreen. The meaning of the event and the accompanying item passed as a parameter to the event are implementation defined.

*NotifyViews* first calls all the document's child documents' *NotifyViews* functions, which are called with the same parameters. Once all the children have been called, *NotifyViews* passes the event and item to all of the document's associated views. *NotifyViews* returns a **bool**. If any child document or associated view returns **false**, *NotifyViews* returns **false**. Otherwise *NotifyViews* returns **true**.

*QueryViews* sends an event and accompanying parameter just like *NotifyViews*. The difference is that, whereas *NotifyViews* returns **true** when any child or view returns **true**, *QueryViews* returns a pointer to the first view that returns **true**. This lets you find a view that meets some condition and then perform some action on that view. If no views return **true**, *QueryViews* returns 0.

Another difference between *NotifyViews* and *QueryViews* is that *NotifyViews* always sends the event and its parameter to *all* children and associated views, whereas *QueryViews* stops at the first view that returns **true**.

For example, suppose you have a document class that contains graphics data in a bitmap. You want to know which of your associated views is displaying a certain area of the current bitmap. You can define an event such as WM_CHECKRECT. Then you can set up a *TRect* structure containing the coordinates of the rectangle you want to check for. The excerpted code for this would look something like this:

```
DEFINE_RESPONSE_TABLE1(TMyView, TView)
   ⋮
  EV_OWLNOTIFY(WM_CHECKREST, EvCheckRest),
   ⋮
END_RESPONSE_TABLE;

void
MyDocClass::Function()
{
  // Set up a TRect * with the coordinates you want to send.
  TRect *rect = new TRect(100, 100, 300, 300);

  // QueryViews
  TView *view = QueryViews(WM_CHECKRECT, (long) rect);

  // Clear all changes from the view
  if(view)
    view->Clear();
}

// The view response function gets the pointer to the rectangle
// as the long parameter to its response function.
bool
TMyView::EvCheckRest(long item)
{
  TRect *rect = (TRect *) item;
```

```
    // Check to see if rect is equal to this view's.
    if(*rect == this->rect)
      return true;
    else
      return false;
  }
```

You can also set up your own event macros to handle view notifications. See page 136.

# Creating a view class

The user almost never interacts directly with a document. Instead the user works with an interface object, such as a window, a dialog box, or whatever type of display is appropriate for the data being presented and the method in which it is presented. But this interface object doesn't stand on its own. A window knows nothing about the data it displays, the document that contains that data, or about how the user can manipulate and change the data. All this functionality is handled by the view object.

A view forms an interface between an interface object (which can only do what it's told to do) and a document (which doesn't know how to tell the interface object what to do). The view's job is to bridge the gap between the two objects, reading the data from the document object and telling the interface object how to display that data.

This section discusses how to write a view class to work with your document classes.

## Constructing TView

You cannot directly create an instance of *TView*. *TView* contains a number of pure **virtual** functions and placeholder functions whose functionality must be provided in any derived classes. But you must call the *TView* constructor when you are constructing your *TView*-derived object. The *TView* constructor takes one parameter, a reference to the view's associated document. You must provide a valid reference to a *TDocument*-derived object.

## Adding functionality to views

*TView* contains some pure **virtual** functions that you must provide in every new view class. It also contains a few placeholder functions that have no base class functionality. You need to provide new versions of these functions if you plan to use them for anything.

Much like *TDocument*, you should not override a *TView* function unless that function is a virtual. When functions in *TDocument* call functions in your view, they address the view object as a *TView*. If you override a nonvirtual function and the document calls that function, the document actually calls the *TView* version of that function, rendering your function useless in that context.

### TView virtual functions

The following functions are declared **virtual** so you can override them to provide some useful functionality. But most are not declared as pure **virtuals**; you are not *required* to override them to construct a view. Instead, you need to override these functions only if you plan to view them.

*GetViewName* returns the static name of the view. This function is declared as a *pure* **virtual** function; you *must* provide a definition of this function in your view class.

*GetWindow* returns a *TWindow* * that should reference the view's associated interface object if it has one; otherwise, *GetWindow* returns 0.

*SetDocTitle* sets the view window's caption. It should be set to call the *SetDocTitle* function in the interface object.

### Adding a menu

*TView* contains the *TMenuDescr* * data member *ViewMenu*. You can assign any existing *TMenuDescr* object to this member. The menu should normally be set up in the view's constructor. This menu is then merged with the frame window's menu when the view is activated.

## Adding a display to a view

*TView* itself makes no provision for displaying data—it has no pointer to a window, no graphics functions, no text display functions, and no keyboard handling. You need to provide this functionality in your derived classes; you can use one of the following methods to do so:

- Add a pointer to an interface object in your derived view class.
- Mix in the functionality of an interface object with that of *TView* when deriving your new view class.

Each of these methods has its advantages and drawbacks, which are discussed in the following sections. You should weigh the pros and cons of each approach before deciding how to build your view class.

### Adding pointers to interface objects

To add a pointer to an interface object to your *TView*-derived class, add the member to the new class and instantiate the object in the view class' constructor. Access to the interface object's data and function members is through the pointer.

The advantage of this method is that it lets you easily attach and detach different interface objects. It also lets you use different types of interface objects by making the pointer a pointer to a common base class of the different objects you might want to use. For example, you can use most kinds of interface objects by making the pointer a *TWindow* *.

The disadvantage of this method is that event handling must go through either the interface object or the application first. This basically forces you to either use a derived interface object class to add your own event-handling functions that make reference to

the view object, or handle the events through the application object. Either way, you decrease your flexibility in handling events.

### Mixing TView with interface objects

Mixing *TView* or a *TView*-derived object with an interface object class gives you the ability to display data from a document, and makes that ability integral with handling the flow of data to and from the document object. To mix a view class with an interface object class is a fairly straightforward task, but one that must be undertaken with care.

To derive your new class, define the class based on your base view class (*TView* or a *TView*-derived class) and the selected interface object. The new constructor should call the constructors for both base classes, and initialize any data that needs to be set up. At a bare minimum, the new class must define any functions that are declared pure **virtual** in the base classes. It should also define functions for whatever specialized screen activities it needs to perform, and define event-handling functions to communicate with both the interface element and the document object.

The advantage of this approach is that the resulting view is highly integrated. Event handling is performed in a central location, reducing the need for event handling at the application level. Control of the interface elements does not go through a pointer but is also integrated into the new view class.

However, if you use this approach, you lose the flexibility you have with a pointer. You cannot quickly detach and attach new interface objects; the interface object is an organic part of the whole view object. You also cannot exchange different types of objects by using a base pointer to a different interface object classes. Your new view class is locked into a single type of interface element.

## Closing a view

Like most other objects, *TView* provides functions that let you safely close and destroy the object.

*~TView* does fairly little. It calls its associated document's *DetachView* function, thus removing itself from the document's list of views.

*TView* also provides a *CanClose* function, which calls its associated document's *CanClose* function. Therefore the view's ability to close depends on the document's ability to close.

# Doc/View event handling

You should normally handle Doc/View events through both the application object and your view's interface element. You can either control the view's display through a pointer to an interface object or mix the functionality of the interface object with a view class (see page 135 for details on constructing an interface element).

You can find more information about event handling and response tables in an ObjectWindows application in Chapter 4.

## Doc/View event handling in the application object

The application object generally handles only a few events, indicating when a document or a view has been created or destroyed. The *dnCreate* event is posted whenever a view or document is created. The *dnClose* event is posted whenever a view or document is closed.

To set up response table entries for these events, add the EV_OWLDOCUMENT and EV_OWLVIEW macros to your response table:

- Use the EV_OWLDOCUMENT macro to check for:
  - The *dnCreate* event when a new document object is created. The standard name used for the handler function is *EvNewDocument*. *EvNewDocument* takes a reference to the new *TDocument*-derived object and returns **void**.
  - The *dnClose* event when a document object is about to be closed. The standard name used for the handler function is *EvCloseDocument*. *EvCloseDocument* takes a reference to the *TDocument*-derived object that is being closed and returns **void**.

  The response table entries and function declarations for these two macros would look like this:

```
DEFINE_RESPONSE_TABLE1(MyDVApp, TApplication)
   ⋮
  EV_OWLDOCUMENT(dnCreate, EvNewDocument),
  EV_OWLDOCUMENT(dnClose, EvCloseDocument),
   ⋮
END_RESPONSE_TABLE;

void EvNewDocument(TDocument& document);
void EvCloseDocument(TDocument& document);
```

- Use the EV_OWLVIEW macro to check for:
  - The *dnCreate* event when a new view object is constructed. The standard name used for the handler function is *EvNewView*. *EvNewView* takes a reference to the new *TView*-derived object and returns **void**.

    If the view contains a window interface element, either by inheritance or through a pointer, the interface element typically has not been created when the view is constructed. You can then modify the interface element's creation attributes before actually calling the *Create* function.
  - The *dnClose* event when a view object is destroyed. The standard name used for the handler function is *EvCloseView*. *EvCloseView* takes a reference to the *TView*-derived object that is being destroyed and returns **void**.

  The response table entries and function declarations for these two macros would look like this:

```
DEFINE_RESPONSE_TABLE1(MyDVApp, TApplication)
   ⋮
  EV_OWLVIEW(dnCreate, EvNewView),
  EV_OWLVIEW(dnClose,  EvCloseView),
   ⋮
```

```
END_RESPONSE_TABLE;

void EvNewView(TView &view);
void EvCloseView(TView &view);
```

## Doc/View event handling in a view

The header file docview.h provides a number of response table macros for predefined events, along with the handler function names and type checking for the function declarations. You can also define your own events and functions to handle those events using the NOTIFY_SIG and VN_DEFINE macros.

### Handling predefined Doc/View events

There are a number of predefined Doc/View events. Each event has a corresponding response table macro and handler function signature defined. Note that the Doc/View model doesn't provide versions of these functions. You must declare the functions in your view class and provide the appropriate functionality for each function.

**Table 10.3** Predefined Doc/View event handlers

| Response table macro | Event name | Event handler | Event |
|---|---|---|---|
| EV_VN_VIEWOPENED | *vnViewOpened* | *VnViewOpened*(*TView* *) | Indicates that a new view has been constructed. |
| EV_VN_VIEWCLOSED | *vnViewClosed* | *VnViewClosed*(*TView* *) | Indicates that a view is about to be destroyed. |
| EV_VN_DOCOPENED | *vnDocOpened* | *VnDocOpened*(**int**) | Indicates that a new document has been opened. |
| EV_VN_DOCCLOSED | *vnDocClosed* | *VnDocClosed*(**int**) | Indicates that a document has been closed. |
| EV_VN_COMMIT | *vnCommit* | *VnCommit*(**bool**) | Indicates that changes made to the data in the view should be committed to the document. |
| EV_VN_REVERT | *vnRevert* | *VnRevert*(**bool**) | Indicates that changes made to the data in the view should be discarded and the data should be restored from the document. |
| EV_VN_ISDIRTY | *vnIsDirty* | *VnIsDirty*(**void**) | Should return **true** if changes have been made to the data in the view and not yet committed to the document, otherwise returns **false**. |
| EV_VN_ISWINDOW | *vnIsWindow* | *VnIsWindow*(HWND) | Should return **true** if the HWND parameter is the same as that of the view's display window. |

All the event-handling functions used for these messages return **bool**.

### Adding custom view events

You can use the VN_DEFINE and NOTIFY_SIG macros to post your own custom view events and to define corresponding response table macros and event-handling functions. This section describes how to define an event and set up the event-handling function and response table macro for that event.

First you must define the name of the event you want to handle. By convention, this name should begin with the letters *vn* followed by the event name. A custom view event should be defined as a **const int** greater than the value *vnCustomBase*. You can define your event values as being *vnCustomBase* plus some offset value. For example, suppose you are defining an event called *vnPenChange*. The code would look something like this:

```
const int vnPenChange = vnCustomBase + 1;
```

Next use the NOTIFY_SIG macro to specify the signature of the event-handling function. The NOTIFY_SIG macro takes two parameters, the first being the event name and the second being the exact parameter type to be passed to the function. The size of this parameter can be no larger than type **long**; if the object being passed is larger than a **long**, you must pass it by pointer. For example, suppose for the *vnPenChange* event, you want to pass a *TPen* object to the event-handling function. Because a *TPen* object is quite a bit larger than a **long**, you must pass the object by pointer. The macro would look something like this:

```
NOTIFY_SIG(vnPenChange, TPen *)
```

Now you need to define the response table macro for your event. By convention, the macro name uses the event name, in all uppercase letters, preceded by EV_VN_. Use the **#define** macro to define the macro name. Use the VN_DEFINE macro to define the macro itself. This macro takes three parameters:

- Event name
- Event-handling function name (by convention, the same as the event name preceded by *Vn* instead of the *vn* used for the event name)
- Size of the parameter for the event-handling function; this can have four different values:
  - void
  - int (size of an int parameter depends on the platform)
  - long (32-bit integer or far pointer)
  - pointer (size of a pointer parameter depends on the memory model)

  You should specify the value that most closely corresponds to the event-handling function's parameter type.

The definition of the response table macro for the *vnPenChange* event would look something like this:

```
#define EV_VN_PENCHANGE \
  VN_define (vnPenChange, VnPenChange, pointer)
```

Note that the third parameter of the VN_DEFINE macro in this case is pointer. This indicates the size of the value passed to the event-handling function.

## Doc/View properties

Every document and view object contains a list of properties, along with functions you can use to query and change those properties. The properties contain information about the object and its capabilities. When the document manager creates or destroys a document or view object, it sends a notification event to the application. The application can query the object's properties to determine how to proceed. Views can also access the properties of their associated document.

## Property values and names

*TDocument* and *TView* each have some general properties. These properties are available in any classes derived from *TDocument* and *TView*. These properties are indexed by a list of enumerated values. The first property for every *TDocument*- and *TView*-derived class should be *PrevProperty*. The last value in the property list should be *NextProperty*. These two values delimit the property list of every document and view object; they ensure that your property list starts at the correct value and doesn't overstep another property's value, and allows derived classes to ensure that their property lists start at a suitable value. *PrevProperty* should be set to the value of the most direct base class' *NextProperty* – 1.

For example, a property list for a class derived from *TDocument* might look something like this:

```
enum
{
  PrevProperty = TDocument::NextProperty-1,
  Size,
  StorageSize,
  NextProperty,
};
```

Note the use of the scope operator (::) when setting *PrevProperty*. This ensures that you set *PrevProperty* to the correct value for *NextProperty*.

Property names are usually contained in an array of strings, with the position of each name in the array corresponding to its enumerated property index. But, when adding properties to a derived class, you can store and access the strings in whatever style you want. Because you have to write the functions to access the properties, complicated storage schemes aren't recommended. A property name should be a simple description of the property.

Property attributes are likewise usually contained in an array, this time an array on **ints**. Again, you can handle this however you like. But the usual practice is to have the attributes for a property contained in an array corresponding to the value of its property index. The attributes indicate how the property can be accessed:

**Table 10.4** Doc/View property attributes

| Attribute | Function |
|---|---|
| pfGetText | Property accessible as text format. |
| pfGetBinary | Property accessible as native non-text format. |
| pfConstant | Property cannot be changed once the object is created. |
| pfSettable | Property settable, must supply native format. |
| pfUnknown | Property defined but unavailable in **this** object. |
| pfHidden | Property should be hidden from normal browse (don't let the user see its name or value). |
| pfUserDef | Property has been user-defined at run time. |

## Accessing property information

There are a number of functions provided in both *TDocument* and *TView* for accessing Doc/View object property information. All of these functions are declared virtual. Because the property access functions are virtual, the function in the most derived class gets called first, and can override properties defined in a base class. It's the responsibility of each class to implement property access and to resolve its property names.

You normally access a property by its index number. Use the *FindProperty* function with the property name. *FindProperty* takes a **char** * parameter and searches the property list for a property with the same name. It returns an **int**, which is used as the property index for succeeding calls.

You can also use the *PropertyName* function to find the property name frpm the index. *PropertyName* takes an **int** parameter and returns a **char** * containing the name of the property.

You can get the attributes of a property using the *PropertyFlags* function. This function takes an **int** parameter, which should be the index of the desired property, and returns an **int**. You can determine whether a flag is set by using the **&** operator. For example, to determine whether you can get a property value in text form, you should check to see whether the *pfGetText* flag is set:

```
if(doc->PropertyFlags() & pfGetText)
{
  // Get property as text....
}
```

### Getting and setting properties

You can use the *GetProperty* and *SetProperty* functions to query and modify the values of a Doc/View object's properties.

The *GetProperty* function lets you find out the value of a property:

```
int GetProperty(int index, void far* dest, int textlen = 0);
```

where:

- *index* is the property index.
- *dest* is used by *GetProperty* to contain the property data.
- *textlen* indicates the size of the memory array pointed to by *dest*. If *textlen* is 0, the property data is returned in binary form; otherwise the data is returned in text form. Data can be returned in binary form only if the *pfGetBinary* attribute is set; it can be returned in text form only if the *pfGetText* attribute is set. To get or set the binary data of properties, the data type and the semantics must be known by the caller.

The *SetProperty* function lets you set the value of a property:

```
bool SetProperty(int index, const void far* src)
```

where:

- *index* is the property index.

- *src* contains the data to which the property should be set; *src* must be in the correct native format for the property.

A derived class that duplicates property names should provide the same behavior and data type.

Chapter

11

# Control objects

Windows provides a number of *controls*, which are standard user-interface elements with specialized behavior. ObjectWindows provides several *custom controls*; it also provides interface objects for controls so you can use them in your applications. Interface objects for controls are called *control objects*.

To learn more about interface objects, see Chapter 3. This chapter covers the following topics:

- Tasks common to all control objects
  - Constructing and destroying control objects
  - Communicating with control objects
- Using each of the different control objects
- Setting and reading control values

## Control classes

The following table lists all the control classes ObjectWindows provides.

**Table 11.1** Controls and their ObjectWindows classes

| Control | Class name | Description |
|---|---|---|
| **Standard Windows controls:** | | |
| List box | *TListBox* | A list of items to choose from. |
| Scroll bar | *TScrollBar* | A scroll bar (like those in scrolling windows and list boxes) with direction arrows and an elevator thumb. |
| Button | *TButton* | A button with an associated text label. |
| Check box | *TCheckBox* | A button consisting of a box that can be checked (on) or unchecked (off), with an associated text label. |
| Radio button | *TRadioButton* | A button that can be checked (on) or unchecked (off), usually in mutually exclusive groups. |
| Group box | *TGroupBox* | A static rectangle with optional text in the upper-left corner. |

Table 11.1 Controls and their ObjectWindows classes (continued)

| Control | Class name | Description |
|---|---|---|
| Edit control | *TEdit* | A field for the user to type text in. |
| Static control | *TStatic* | Visible text the user can't change. |
| Combo box | *TComboBox* | A combined list box and edit or static control. |
| **Custom ObjectWindows controls:** | | |
| Slider | *THSlider* and *TVSlider* | Horizontal and vertical controls that let the user choose from an upper and lower range (similar to scroll bars). |
| Gauge | *TGauge* | Static controls that display a range of process completion. |

Control object example programs can be found in EXAMPLES\OWL\OWLAPI and EXAMPLES\OWL\OWLAPPS.

## What are control objects?

To Windows, controls are just specialized windows. In ObjectWindows, *TControl* is derived from *TWindow*. Control objects and window objects are similar in how they behave as child windows, and in how you create and destroy them. Standard controls differ from other windows, however, in that Windows handles their event messages and is responsible for painting them. Custom ObjectWindows controls handle these tasks themselves because the ObjectWindows control classes contain the code needed to paint the controls and handle events.

In many cases, you can directly use instances of the classes listed in the previous table. However, sometimes you might need to create derived classes for specialized behavior. For example, you might derive a specialized list box class from *TListBox* called *TFontListBox* that holds the names of all the fonts available to your application and automatically displays them when you create an instance of the class.

# Constructing and destroying control objects

Regardless of the type of control object you're using, there are several tasks you need to perform for each:

- Constructing the control object
- Showing the control
- Destroying the control

## Constructing control objects

Constructing a control object is no different from constructing any other child window. Generally, the parent window's constructor calls the constructors of all its child windows. Notifications are described in Chapter 3. Controls communicate with parent windows in special ways (called *notifications*) in addition to the usual links between parent and child.

To construct and initialize a control object:

1. Add a control object pointer data member to the parent window.
2. Call the control object's constructor.
3. Change any control attributes.
4. Initialize the control in *SetupWindow*.

Each of these steps is described in the following sections.

## Adding the control object pointer data member

Often when you construct a control in a window, you want to keep a pointer to the control in a window object data member. This is for convenience in accessing the control's member functions. Here's a fragment of a parent window object with the declaration for a pointer to a button control object:

```
class TMyWindow : public TWindow
{
  TButton *OkButton;
  :
};
```

Controls that you rarely manipulate, like static text and group boxes, don't need these pointer data members. The following example constructs a group box without a data member and a button with a data member (*OkButton*):

```
TMyWindow::TMyWindow(TWindow *parent, const char far *title)
  : TWindow(parent, title)
{
  new TGroupBox(this, ID_GROUPBOX, "Group box", 10, 10, 100, 100);
  OkButton = new TButton(this, IDOK, "OK", 10, 200, 50, 50, true);
}
```

## Calling control object constructors

Some control object constructors are passed parameters that specify characteristics of the control object. These parameters include

- A pointer to the parent window object
- A resource identifier
- The x-coordinate of the upper-left corner
- The y-coordinate of the upper-left corner
- The width
- The height
- Optional module pointer

For example, one of *TListBox*'s constructors is declared as follows:

```
TListBox(TWindow *parent, int resourceId,
         int x, int y, int w, int h,
         TModule* module = 0);
```

There are also constructors for associating a control object with an interface element (for example a dialog box) created from a resource definition:

```
TListBox(TWindow* parent, int resourceId, TModule* module = 0);
```

### Changing control attributes

All control objects get the default window styles WS_CHILD, WS_VISIBLE, WS_GROUP, and WS_TABSTOP. If you want to change a control's style, you manipulate its *Attr.Style*, as described in Chapter 7. Each control type also has other styles that define its particular properties.

Each control object inherits certain window styles from its base classes. You should rarely assign a value to *Attr.Style*. Instead, you should use the bitwise assignment operators (|= and &=) to "mask" in or out the window style you want. For example:

```
// Mask in the WS_BORDER window style
Attr.Style |= WS_BORDER;

// Mask out the WS_VSCROLL style
Attr.Style &= ~WS_VSCROLL;
```

Using the bitwise assignment operators helps ensure that you don't inadvertently remove a style.

### Initializing the control

A control object's interface element is automatically created by the *SetupWindow* member function inherited by the parent window object. Make sure that when you derive new window classes, you call the base class' *SetupWindow* member function before attempting to manipulate its controls (for example, by calling control object member functions, sending messages to those controls, and so on).

You must not initialize controls in their parent window object's constructor. At that time, the controls' interface elements haven't yet been created.

Here's a typical *SetupWindow* function:

```
void
TMyWindow::SetupWindow()
{
  TWindow::SetupWindow(); // Lets TWindow create any child controls

  list1->AddString("Item 1");
  list1->AddString("Item 2");
}
```

## Showing controls

It's not necessary to call the Windows function *Show* to display controls. Controls are child windows, and Windows automatically displays and repaints them along with the parent window. You can use *Show*, however, to hide or reveal controls on demand.

## Destroying the control

Destroying controls is the parent window's responsibility. The control's interface element is automatically destroyed along with the parent window when the user closes

the window or application. The parent window's destructor automatically destroys its child window objects (including child control objects).

# Communicating with control objects

Communication between a window object and its control objects is similar in some ways to the communication between a dialog box object and its controls. Like a dialog box, a window needs a mechanism for manipulating its controls and for responding to control events, such as a list box selection.

## Manipulating controls

One way dialog boxes manipulate their controls is by sending them messages using member functions inherited from *TWindow* (see Chapter 7), with a control message like LB_ADDSTRING. Control objects greatly simplify this process by providing member functions that send control messages for you. *TListBox::AddString*, for example, takes a string as its parameter and adds it to the list box by calling the list box object's *HandleMessage* member function:

```
int
TListBox::AddString(const char far* str)
{
  return (int)HandleMessage(LB_ADDSTRING, 0, (LPARAM)str);
}
```

This example shows how you can call the control objects' member functions via a pointer:

```
ListBox1->AddString("Atlantic City"); //where ListBox1 is a TListBox *
```

## Responding to controls

When a user interacts with a control, Windows sends various control messages. To learn how to respond to control messages, see Chapter 3.

## Making a window act like a dialog box

A dialog box lets the user use the **Tab** key to cycle through all of the dialog box's controls. It also lets the user use the arrow keys to select radio buttons in a group box. To emulate this keyboard interface for windows with controls, call *EnableKBHandler* in the window object's constructor.

# Using particular controls

Each type of control operates somewhat differently from the others. In this section, you'll find specific information on how to use the objects for each of the standard Windows controls and the custom controls supplied with ObjectWindows.

## Using list box controls

Using a list box is the simplest way to ask the user to pick something from a list. The *TListBox* class encapsulates list boxes. *TListBox* defines member functions for:

- Creating list boxes
- Modifying the list of items
- Inquiring about the list of items
- Finding out which item the user selected

### Constructing list box objects

One of *TListBox*'s constructors takes seven parameters: a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an module pointer:

```
TListBox(TWindow *parent,
         int resourceId,
         int x, int y, int w, int h,
         TModule* module = 0);
```

*TListBox* gets the default control styles (WS_CHILD, WS_VISIBLE, WS_GROUP, and WS_TABSTOP; see page 146) and adds LBS_STANDARD, which is a combination of LBS_NOTIFY (to receive notification messages), WS_VSCROLL (to have a vertical scroll bar), LBS_SORT (to sort the list items alphabetically), and WS_BORDER (to have a border). If you want a different list box style, you can modify *Attr.Style* in the list box object's constructor or in its parent's constructor. For example, for a list box that doesn't sort its items, use the following code:

```
listbox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
listbox->Attr.Style &= ~LBS_SORT;
```

### Modifying list boxes

After you create a list box, you need to fill it with list items (which must be strings). Later, you can add, insert, or remove items or clear the listcompletely. The following table summarizes the member functions you use to perform these actions.

**Table 11.2** TListBox member functions for modifying list boxes

| Member function | Description |
|---|---|
| *ClearList* | Delete every item. |
| *DirectoryList* | Put file names in the list. |
| *AddString* | Add an item. |
| *InsertString* | Insert an item. |
| *DeleteString* | Delete an item. |
| *SetSelIndex*, *SetSel*, or *SetSelString* | Select an item. |
| *SetSelStrings*, *SetSelIndexes*, or *SetSelItemRange* | Select multiple items. |
| *SetTopIndex* | Scroll the list box so the specified item is visible. |
| *SetTabStops* | Set tab stops for multicolumn list boxes. |
| *SetHorizontalExtent* | Set number of pixels by which the list box can scroll horizontally. |
| *SetColumnWidth* | Set width of all columns in multicolumn list boxes. |

**Table 11.2** TListBox member functions for modifying list boxes (continued)

| Member function | Description |
|---|---|
| *SetCaretIndex* | Set index of the currently focused item. |
| *SetItemData* | Set a *uint32* value to be associated with the specified index. |
| *SetItemHeight* | Set the height of item at the specified index or height of all items. |

## Querying list boxes

There are several member functions you can call to find out information about the list box or its item list. The following table summarizes the list box query member functions.

**Table 11.3** TListBox member functions for querying list boxes

| Member functions | Description |
|---|---|
| *GetCount* | Number of items in the list. |
| *FindString* or *FindExactString* | Find string index. |
| *GetTopIndex* | Index of the item at the top of the list box. |
| *GetCaretIndex* | Index of the currently focused item. |
| *GetHorizontalExtent* | Number of pixels the list box can scroll horizontally. |
| *GetItemData* | *uint32* data set by *SetItemData*. |
| *GetItemHeight* | Height, in pixels, of the specified item. |
| *GetItemRect* | Rectangle used to display the specified item. |
| *GetSelCount* | Number of selected items (either 0 or 1). |
| *GetSelIndex* or *GetSel* | Index of the selected item. |
| *GetSelString* | Selected item. |
| *GetSelStrings* or *GetSelIndexes* | Selected items. |
| *GetString* | Item at a particular index. |
| *GetStringLen* | Length of a particular item. |

## Responding to list boxes

The member functions for modifying and querying list boxes let you set values or find out the status of the control at any given time. To know what a user is doing to a list box at run time, however, you have to respond to notification messages from the control.

There are only a few things a user can do with a list box: scroll through the list, click an item, and double-click an item. When the user does one of these things, Windows sends a *list box notification* message to the list box's parent window. Normally, you define notification-response member functions in the parent window object to handle notifications for each of the parent's controls.

The following table summarizes the most common list box notifications:

**Table 11.4** List box notification messages

| Event response table macro | Description |
|---|---|
| EV_LBN_SELCHANGE | An item has been selected with a single mouse click. |
| EV_LBN_DBLCLK | An item has been selected with a double mouse click. |

**Table 11.4** List box notification messages (continued)

| Event response table macro | Description |
|---|---|
| EV_LBN_SELCANCEL | The user has deselected an item. |
| EV_LBN_SETFOCUS | The user has given the list box the focus by clicking or double-clicking an item, or by using *Tab*. Precedes LBN_SELCHANGE notification. |
| EV_LBN_KILLFOCUS | The user has removed the focus from the list box by clicking another control or pressing *Tab*. |

Here's a sample parent window object member function to handle an LBN_SELCHANGE notification:

```
DEFINE_RESPONSE_TABLE1(TLBoxWindow, TFrameWindow)
  EV_LBN_SELCHANGE(ID_LISTBOX, EvListBoxSelChange),
END_RESPONSE_TABLE;

void
TLBoxWindow::EvListBoxSelChange()
{
  int index = ListBox->GetSelIndex();
  if (ListBox->GetStringLen(index) < 10) {
    char string[10];
    ListBox->GetSelString(string, sizeof(string));
    MessageBox(string, "You selected:", MB_OK);
  }
}
```

# Using static controls

Static controls are usually unchanging units of text or simple graphics. The user doesn't interact with static controls, although your application can change the static control's text. See EXAMPLES\OWL\OWLAPI\STATIC for an example showing static controls.

## Constructing static control objects

Because the user never interacts directly with a static control, the application doesn't receive control-notification messages from static controls. Therefore, you can construct most static controls with –1 as the control ID. However, if you want to use *TWindow::SendDlgItemMessage* to manipulate the static control, you need a unique ID.

One of *TStatic*'s constructors is declared as follows:

```
TStatic(TWindow* parent,
        int resourceId,
        const char far* title,
        int x, int y, int w, int h,
        UINT textLen = 0,
        TModule* module = 0);
```

It takes the seven parameters commonly found in this form of a control object constructor (a parent window, a resource ID, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer), and two parameters specific to static controls: the text

string the static control displays and its maximum length (including the terminating NULL). A typical call to construct a static control looks like this:

```
new TStatic(this, -1, "Sample &Text", 170, 20, 200, 24);
```

If you want to be able to change the static control's text, you need to assign the control object to a data member in the parent window object so you can call the static control object's member function. If the static control's text doesn't need to change, you don't need a data member.

*TStatic* gets the default control styles (WS_CHILD, WS_VISIBLE, WS_GROUP, and WS_TABSTOP; see page 146), adds SS_LEFT (to left-align the text), and removes the WS_TABSTOP style (to prevent the user from selecting the control using ***Tab***). To change the style, modify *Attr.Style* in the static control object's constructor. For example, the following code centers the control's text:

```
Attr.Style = (Attr.Style & ~SS_LEFT) | SS_CENTER;
```

To indicate a mnemonic for a nearby control, you can underline one or more characters in the static control's text string. To do this, insert an ampersand `&` in the string immediately preceding the character you want underlined. For example, to underline the `T` in `Text`, use `&Text`. If you want to use an ampersand in the string, use the static style SS_NOPREFIX.

### Modifying static controls

*TStatic* has two member functions for altering the text of a static control: *SetText* sets the text to the passed string, and *Clear* erases the text. You can't change the text of static controls created with the SS_SIMPLE style.

### Querying static controls

*TStatic::GetTextLen* returns the length of the static control's text. To get the text itself, use *TStatic::GetText*.

## Using button controls

Buttons (sometimes called push buttons or command buttons) perform a task each time the button is pressed. There are two kinds of buttons: default buttons and non-default buttons. A default button, distinguished by the button style BS_DEFPUSHBUTTON, has a bold border that indicates the default user response. Nondefault buttons have the button style BS_PUSHBUTTON.

See EXAMPLES\OWL\OWLAPI\BWCC for an example of button controls.

### Constructing buttons

One of *TButton*'s constructors takes the seven parameters commonly found in a control object constructor (a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer), plus a text string that specifies the button's label, and a **bool** flag that indicates whether the button should be a default button. Here's the constructor declaration:

```
TButton(TWindow *parent,
        int resourceId,
        const char far *text,
        int X, int Y, int W, int H,
        bool isDefault = false,
        TModule *module = 0);
```

A typical button would be constructed like this:

```
btn = new TButton(this, ID_BUTTON, "DO_IT!", 38, 48, 316, 24, true);
```

### Responding to buttons

When the user clicks a button, the button's parent window receives a notification message. If the parent window object intercepts the message, it can respond to these events by displaying a dialog box, saving a file, and so on.

To intercept and respond to button messages, define a command response member function for the button. The following example uses ID ID_BUTTON to handle the response to the user clicking the button:

```
DEFINE_RESPONSE_TABLE1(TTestWindow, TFrameWindow)
  EV_COMMAND(ID_BUTTON, HandleButtonMsg),
END_RESPONSE_TABLE;

void
TTestWindow::HandleButtonMsg()
{
  // Button was pressed
}
```

## Using check box and radio button controls

A *check box* generally presents the user with a two-state option. The user can check or uncheck the control, or leave it as is. In a group of check boxes, any or all might be checked. For example, you might use a check box to enable or disable the use of sound in your application.

*Radio buttons*, on the other hand, are used for selecting one of several mutually exclusive options. For example, you might use radio buttons to choose between a number of sounds in your application.

*TCheckBox* is derived from *TButton* and represents check boxes. Since radio buttons share some behavior with check boxes, *TRadioButton* is derived from *TCheckBox*.

Check boxes and radio buttons are sometimes collectively referred to as *selection boxes*. While displayed on the screen, a selection box is either checked or unchecked. When the user clicks a selection box, it's an event, generating a Windows notification. As with other controls, the selection box's parent window usually intercepts and acts on these notifications.

See EXAMPLES\OWL\OWLAPI\BUTTON for radio button and check box control examples.

## Constructing check boxes and radio buttons

*TCheckBox* and *TRadioButton* each have a constructor that takes the seven parameters commonly found in a control object constructor (a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer). They also take a text string and a pointer to a group box object that groups the selection boxes. If the group box object pointer is zero, the selection box isn't part of a group box. Here are one each of their constructors:

```
TCheckBox(TWindow *parent,
          int resourceId,
          const char far *title,
          int x, int y, int w, int h,
          TGroupBox *group = 0,
          TModule *module = 0);

TRadioButton(TWindow *parent,
             int resourceId,
             const char far *title,
             int x, int y, int w, int h,
             TGroupBox *group = 0,
             TModule *module = 0);
```

The following listing shows some typical constructor calls for selection boxes.

```
CheckBox = new TCheckBox(this, ID_CHECKBOX, "Check Box Text", 158, 12, 150, 26);
GroupBox = new TGroupBox(this, ID_GROUPBOX, "Group Box", 158, 102, 176, 108);

RButton1 = new TRadioButton(this, ID_RBUTTON1, "Radio Button 1",
                            174, 128, 138, 24, GroupBox);
RButton2 = new TRadioButton(this, ID_RBUTTON2, "Radio Button 2",
                            174, 162, 138, 24, GroupBox);
```

Check boxes by default have the BS_AUTOCHECKBOX style, which means that Windows handles a click on the check box by toggling the check box. Without BS_AUTOCHECKBOX, you'd have to set the check box's state manually. Radio buttons by default have the BS_AUTORADIOBUTTON style, which means that Windows handles a click on the radio button by checking the radio button and unchecking the other radio buttons in the group. Without BS_AUTORADIOBUTTON, you'd have to intercept the radio button's notification messages and do this work yourself.

## Modifying selection boxes

Checking and unchecking a selection box seems like a job for the application user, not your application. But in some cases, your application needs control over a selection box's state. For example, if the user opens a text file, you might want to automatically check a check box labeled "Save as ANSI text." *TCheckBox* defines several member functions for modifying a check box's state:

**Table 11.5** TCheckBox member functions for modifying selection boxes

| Member function | Description |
|---|---|
| *Check* or *SetCheck (BF_CHECKED)* | Check |
| *Uncheck* or *SetCheck (BF_UNCHECKED)* | Uncheck |

**Table 11.5** TCheckBox member functions for modifying selection boxes (continued)

| Member function | Description |
|---|---|
| Toggle | Toggle |
| SetState | Highlight |
| SetStyle | Change the button's style |

When you use these member functions with radio buttons, ObjectWindows ensures that only one radio button per group is checked, as long as the buttons are assigned to a group.

### Querying selection boxes

Querying a selection box is one way to find out and respond to its state. Radio buttons have two states: checked (BF_CHECKED) and unchecked (BF_UNCHECKED). Check boxes can have an additional (and optional) third state: grayed (BF_GRAYED). The following table summarizes the selection-box query member functions.

**Table 11.6** TCheckBox member functions for querying selection boxes

| Member function | Description |
|---|---|
| GetCheck | Return the check state. |
| GetState | Return the check, highlight, or focus state. |

## Using group boxes

In its simplest form, a group box is a labeled static rectangle that visually groups other controls.

### Constructing group boxes

*TGroupBox* has a constructor that takes the seven parameters commonly found in a control object constructor (a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer), and also takes a text string parameter to label the group:

```
TGroupBox(TWindow *parent,
          int resourceId,
          const char far *text,
          int X, int Y, int W, int H,
          TModule *module = 0);
```

### Grouping controls

Usually a group box visually associates a group of other controls; however, it can also logically associate a group of selection boxes (check boxes and radio buttons). This logical group performs the automatic unchecking (BS_AUTOCHECKBOX, BS_AUTORADIOBUTTON) discussed on page 153.

To add a selection box to a group box, pass a pointer to the group box object in the selection box's constructor call.

### Responding to group boxes

When an event occurs that might change the group box's selections (for example, when a user clicks a button or the application calls *Check*), Windows sends a notification message to the group box's parent window. The parent window can intercept the message for the group box as a whole, rather than responding to the individual selection boxes in the group box. To find out which control in the group was affected, you can read the current status of each control.

## Using scroll bars

Scroll bars are the primary mechanism for changing the user's view of an application window, a list box, or a combo box. However, you might want a separate scroll bar to perform a specialized task, such as controlling the temperature on a thermostat or the color in a drawing program. Use *TScrollBar* objects when you need a separate, customizable scroll bar.

See EXAMPLES\OWL\OWLAPI\SCROLLER for a scroll bar control example.

### Constructing scroll bars

*TScrollBar* has a constructor that takes the seven parameters commonly found in a control object constructor (a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer), and also takes a **bool** flag parameter that specifies whether the scroll bar is horizontal. Here's a *TScrollBar* constructor declaration:

```
TScrollBar(TWindow *parent,
           int resourceId,
           int x, int y, int w, int h,
           bool isHScrollBar,
           TModule *module = 0);
```

If you specify a height of zero for a horizontal scroll bar or a width of zero for a vertical scroll bar, Windows gives it a standard height and width. This code creates a standard-height horizontal scroll bar:

```
new TScrollBar(this, ID_THERMOMETER, 100, 150, 180, 0, true);
```

*TScrollBar*'s constructor constructs scroll bars with the style SBS_HORZ for horizontal scroll bars and SBS_VERT for vertical scroll bars. You can specify additional styles, such as SBS_TOPALIGN, by changing the scroll bar object's *Attr.Style*.

### Controlling the scroll bar range

One attribute of a scroll bar is its *range*, which is the set of all possible *thumb* positions. The thumb is the scroll bar's sliding box that the user drags or scrolls. Each position is associated with an integer. The parent window uses this integer, the *position*, to set and query the scroll bar. By default, a scroll bar object's range is 1 to 100.

The thumb's minimum position (at the top of a vertical scroll bar and the left of a horizontal scroll bar) corresponds to position 1, and the thumb's maximum position corresponds to position 100. Use *SetRange* to set the range differently.

## Controlling scroll amounts

A scroll bar has two other important attributes: its *line magnitude* and *page magnitude*. The line magnitude, initialized to 1, is the distance, in range units, the thumb moves when the user clicks the scroll bar's arrows. The page magnitude, initialized to 10, is the distance, also in range units, the thumb moves when the user clicks the scrolling area. You can change these values by changing the *TScrollBar* data members *LineMagnitude* and *PageMagnitude*.

## Querying scroll bars

*TScrollBar* has two member functions for querying scroll bars:

- *GetRange* gets the upper and lower ranges.
- *GetPosition* gets the current thumb position.

## Modifying scroll bars

Modifying scroll bars is usually done by the user, but your application can also modify a scroll bar directly:

- *SetRange* sets the scrolling range.
- *SetPosition* sets the thumb position.
- *DeltaPos* moves the thumb position.

## Responding to scroll-bar messages

When the user moves a scroll bar's thumb or clicks the scroll arrows, Windows sends a scroll bar notification message to the parent window. If you want your window to respond to scrolling events, respond to the notification messages.

Scroll bar notification messages are slightly different from other control notification messages. They're based on the WM_HSCROLL and WM_VSCROLL messages, rather than WM_COMMAND command messages. Therefore, to respond to scroll bar notification messages, you need to define *EvHScroll* or *EvVScroll* event response functions, depending on whether the scroll bar is horizontal or vertical:

```
class TTestWindow : public TFrameWindow
{
  public:
    TTestWindow(TWindow* parent, const char* title);
    virtual void SetupWindow();

    void EvHScroll(UINT code, UINT pos, HWND wnd);

  DECLARE_RESPONSE_TABLE(TTestWindow);
};

DEFINE_RESPONSE_TABLE1(TTestWindow, TFrameWindow)
  EV_WM_HSCROLL,
END_RESPONSE_TABLE;
```

Usually, you respond to all the scroll bar notification messages by retrieving the current thumb position and taking appropriate action. In that case, you can ignore the notification code:

```
void
TTestWindow::EvHScroll(UINT code, UINT pos, HWND wnd)
{
  TFrameWindow::EvHScroll(); // perform default WM_HSCROLL processing
  int newPos = ScrollBar->GetPosition();
  // do some processing with newPos
}
```

### Avoiding thumb tracking messages

You might not want to respond to the scroll bar notification messages while the user is dragging the scroll bar's thumb, because the user is usually dragging the thumb quickly, generating many notification messages. It's more efficient to wait until the user has stopped moving the thumb, and then respond. To do this, screen out the notification messages that have the SB_THUMBTRACK code.

### Specializing scroll bar behavior

You might want a scroll bar object respond to its own notification messages. *TWindow* has built-in support for dispatching scroll bar notification messages back to the scroll bar. *TWindow::EvHScroll* or *TWindow::EvVScroll* execute the appropriate *TScrollBar* member function based on the notification code. For example:

```
class TSpecializedScrollBar : public TScrollBar
{
  public:
    virtual void SBTop();
};

void
TSpecializedScrollBar::SBTop()
{
  TScrollBar::SBTop();
  ::sndPlaySound("AT-TOP.WAV", SND_ASYNC); // play sound
}
```

Be sure to call the base member functions first. They correctly update the scroll bar to its new position.

The following table associates notification messages with the corresponding *TScrollBar* member function:

**Table 11.7** Notification codes and TScrollBar member functions

| Notification message | *TScrollBar* member function |
|---|---|
| SB_LINEUP | *SBLineUp* |
| SB_LINEDOWN | *SBLineDown* |
| SB_PAGEUP | *SBPageUp* |
| SB_PAGEDOWN | *SBPageDown* |

**Table 11.7** Notification codes and TScrollBar member functions (continued)

| Notification message | *TScrollBar* member function |
|---|---|
| SB_THUMBPOSITION | *SBThumbPosition* |
| SB_THUMBTRACK | *SBThumbTrack* |
| SB_TOP | *SBTop* |
| SB_BOTTOM | *SBBottom* |

## Using sliders and gauges

Sliders are specialized scrollers used for nonscrolling position information. The abstract base class *TSlider* is derived from the *TScrollBar* class. Like other control constructors, the *TSlider* constructor takes the seven parameters commonly found in a control object constructor (a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer), and also takes a *TResId* object, which is a bitmap resource identifier. The bitmap is displayed as the thumb knob for the slider. Here's the *TSlider* constructor:

```
THSlider(TWindow* parent,
         int id,
         int X, int Y, int W, int H,
         TResId thumbResId = IDB_HSLIDERTHUMB,
         TModule* module = 0);
```

To implement a class based on *TSlider*, you must implement a number of functions which are declared as pure virtual functions in *TSlider*:

**Table 11.8** Pure virtual functions in TSlider

| Name | Function |
|---|---|
| *HitTest* | Determines whether a point in inside the thumb or any other "hot" area of the slider. |
| *NotifyParent* | Notifies the slider's parent of a scroll event. |
| *PaintRuler* | Paints the ruler. |
| *PaintSlot* | Paints the slot that the thumb rides over. |
| *PointToPos* | Converts a point inside the slider to a slider position. |
| *PosToPoint* | Converts a slider position to a point inside the slider. |
| *SetupThumbRgn* | Sets up the thumb region. By default, this is a simple rectangle. |
| *SlideThumb* | Slides the thumb and does the required blitting and painting. |

Two classes derived from *TSlider*, *THSlider* and *TVSlider*, implement vertical and horizontal slider versions. Both *THSlider* and *TVSlider* have only one constructor. These constructors resemble the *TSlider* constructor, with the exception that each has a default value for the thumb knob bitmap:

```
THSlider(TWindow* parent,
         int id,
         int X, int Y, int W, int H,
         TResId thumbResId = IDB_HSLIDERTHUMB,
         TModule* module = 0);
```

```
TVSlider(TWindow* parent,
         int id,
         int X, int Y, int W, int H,
         TResId thumbResId = IDB_VSLIDERTHUMB,
         TModule* module = 0);
```

Gauges are controls that display duration or other information about an ongoing process. Class *TGauge* implements gauges, and is derived from class *TControl*. The *TGauge* constructor looks like this:

```
TGauge(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 *TGauge* constructor has the normal control constructor parameters (a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer). The *isHorizontal* parameter determines whether you get a horizontal or vertical gauge. If *isHorizontal* is *true*, the gauge is displayed horizontally. If *isHorizontal* is *false*, the gauge is displayed vertically. The default is horizontal. Horizontal gauges are usually used to display process information, and vertical gauges are usually used to display analog information.

The *margin* parameter determines the size of the gauge's margin.

See EXAMPLES\OWL\OWLAPI\SLIDER for slider and gauge control examples.

## Using edit controls

Edit controls are interactive static controls. They're rectangular areas that can be filled with text, modified, and cleared by the user or application. Edit controls are very useful as fields for data entry screens. They support the following operations:

- User text input
- Dynamic display of text (by the application)
- Cutting, copying, and pasting to the Clipboard
- Multiline editing (good for text editors)

See EXAMPLES\OWL\OWLAPI\VALIDATE for an edit controls example.

### Constructing edit controls

One of *TEdit*'s constructors takes parameters for an initial text string, maximum string length (including the terminating NULL), and a **bool** flag specifying whether or not it's a multiline edit control (in addition to the parent window, resource identifier, and placement coordinates). This *TEdit* constructor is declared as follows:

```
TEdit(TWindow *parent,
      int resourceId,
      const char far *text,
      int x, int y, int w, int h,
```

```
        UINT textLen,
        bool multiline = false,
        TModule *module = 0);
```

By default, the edit control has the styles ES_LEFT (for left-aligned text), ES_AUTOHSCROLL (for automatic horizontal scrolling), and WS_BORDER (for a visible border surrounding the edit control). Multiline edit controls get the additional styles ES_MULTILINE (specifies a multiline edit control), ES_AUTOVSCROLL (automatic vertical scrolling), WS_VSCROLL (vertical scroll bar), and WS_HSCROLL (horizontal scroll bar).

The following are typical edit control constructor calls, one for a single-line control, the other multiline:

```
Edit1 = new TEdit(this, ID_EDIT1, "Default Text", 20, 50, 150, 30, MAX_TEXTLEN, false);
Edit2 = new TEdit(this, ID_EDIT2, "", 260, 50, 150, 30, MAX_TEXTLEN, true);
```

## Using the Clipboard and the Edit menu

You can directly transfer text between an edit control object and the Windows Clipboard using *TEdit* member functions. You probably want to give users access to these member functions by giving your window an Edit menu.

Edit control objects have built-in responses to menu items like Edit | Copy and Edit | Undo. *TEdit* has command response member functions, such as *CmEditCopy* and *CmEditUndo*, which ObjectWindows invokes in response to users choosing items from the parent window's Edit menu.

The table below shows the Clipboard and editing member functions and the menu commands that invoke them.

**Table 11.9** TEdit member functions and Edit menu commands

| Member function | Menu command | Description |
|---|---|---|
| *Copy* | CM_EDITCOPY | Copy text to Clipboard. |
| *Cut* | CM_EDITCUT | Cut text to Clipboard. |
| *Undo* | CM_EDITUNDO | Undo last edit. |
| *Paste* | CM_EDITPASTE | Paste text from Clipboard. |
| *DeleteSelection* | CM_EDITDELETE | Delete selected text. |
| *Clear* | CM_EDITCLEAR | Clear entire edit control. |

To add an editing menu to a window that contains edit control objects, define a menu resource for the window using the menu commands listed above. You don't need to write any new member functions.

## Querying edit controls

Often, you want to query an edit control to store the entry for later use. *TEdit* has a number of querying member functions. Many of the edit control query and modification member functions return, or require you to specify, a line number or a character's position in a line. All of these indexes start at zero. In other words, the first

line is line zero and the first character of a line is character zero. The following table summarizes *TEdit*'s query member functions.

**Table 11.10** TEdit member functions for querying edit controls

| Member function | Description |
|---|---|
| *IsModified* | Find out if text has changed. |
| *GetText* | Retrieve all text. |
| *GetLine* | Retrieve a line. |
| *GetNumLines* | Get number of lines. |
| *GetLineLength* | Get length of a given line. |
| *GetSelection* | Get index of selected text. |
| *GetSubText* | Get a range of characters. |
| *GetLineIndex* | Count characters before a line. |
| *GetLineFromPos* | Find the line containing an index. |
| *GetRect* | Get formatting rectangle. |
| *GetHandle* | Get memory handle. |
| *GetFirstVisibleLine* | Get index of first visible line. |
| *GetPasswordChar* | Get character used in passwords. |
| *GetWordBreakProc* | Get word-breaking procedure. |
| *CanUndo* | Find out if edit can be undone. |

Text that spans lines in a multiline edit control contains two extra characters for each line break: a carriage return ('\r') and a line feed ('\n'). *TEdit*'s member functions retain the text's formatting when they return text from a multiline edit control. When you insert this text back into an edit control, paste it from the Clipboard, write it to a file, or print it to a printer, the line breaks appear as they did in the edit control. When you use query member functions to get a specified number of characters, be sure to account for the two extra characters in a line break.

## Modifying edit controls

Many uses of edit controls require that your application explicitly substitute, insert, clear, or select text. *TEdit* supports those operations, plus the ability to force the edit control to scroll.

**Table 11.11** TEdit member functions for modifying edit controls

| Member function | Description |
|---|---|
| *Clear* | Delete all text. |
| *DeleteSelection* | Delete selected text. |
| *DeleteSubText* | Delete a range of characters. |
| *DeleteLine* | Delete a line of text. |
| *Insert* | Insert text. |
| *Paste* | Paste text from Clipboard. |
| *SetText* | Replace all text. |
| *SetSelection* | Select a range of text. |
| *Scroll* | Scroll text. |

**Table 11.11** TEdit member functions for modifying edit controls (continued)

| Member function | Description |
|---|---|
| *ClearModify* | Clear the modified flag. |
| *Search* | Search for text. |
| *SetRect* or *SetRectNP* | Set formatting rectangle. |
| *FormatLines* | Turn on or off soft line breaks. |
| *SetTabStops* | Set tab stops. |
| *SetHandle* | Set local memory handle. |
| *SetPasswordChar* | Set password character. |
| *SetReadOnly* | Make the edit control read-only. |
| *SetWordBreakProc* | Set word-breaking procedure. |
| *EmptyUndoBuffer* | Empty undo buffer. |

# Using combo boxes

A combo box control is a combination of two other controls: a list box and an edit or static control. It serves the same purpose as a list box—it lets the user choose one text item from a scrollable list of text items by clicking the item with the mouse. The edit control, grafted to the top of the list box, provides another selection mechanism, allowing users to type the text of the desired item. If the list box area of the combo box is displayed, the desired item is automatically selected. *TComboBox* is derived from *TListBox* and inherits its member functions for modifying, querying, and selecting list items. In addition, *TComboBox* provides member functions for manipulating the list part of the combo box, which, in some types of combo boxes, can *drop down* on request.

See EXAMPLES\OWL\OWLAPI\COMBOBOX for a combo box control example.

## Varieties of combo boxes

There are three types of combo boxes: simple, drop down, and drop down list. All combo boxes show their edit area at all times, but some can show and hide their list box areas. The following table summarizes the properties of each type of combo box.

**Table 11.12** Summary of combo box styles

| Style | Can hide list? | Text must match list? |
|---|---|---|
| Simple | No | No |
| Drop down | Yes | No |
| Drop down list | Yes | Yes |

From a user's perspective, these are the distinctions between the different styles of combo boxes:

- A simple combo box cannot hide its list box area. Its edit area behaves just like an edit control; the user can enter and edit text, and the text doesn't need to match one of the items in the list. If the text does match, the corresponding list item is selected.
- A drop down combo box behaves like a simple combo box, with one exception. In its initial state, its list area isn't displayed. It appears when the user clicks on the icon to

the right of the edit area. When drop down combo boxes aren't being used, they take up less space than a simple combo box or a list box.

- The list area of a drop down list combo box behaves like the list area of a drop down combo box—it appears only when needed. The two combo box types differ in the behavior of their edit areas. Whereas drop down edit areas behave like regular edit controls, drop down list edit areas are limited to displaying only the text from one of their list items. When the edit text matches the item text, no more characters can be entered.

## Choosing combo box types

Drop down list combo boxes are useful in cases where no other selection is acceptable besides those listed in the list area. For example, when choosing a printer, you can only choose a printer accessible from your system.

On the other hand, drop down combo boxes can accept entries other than those found in the list. A typical use of drop down combo boxes is selecting disk files for opening or saving. The user can either search through directories to find the appropriate file in the list, or type the full path name and file name in the edit area, regardless of whether the file name appears in the list area.

## Constructing combo boxes

*TComboBox* has two constructors. The first constructor takes the seven parameters commonly found in a control object constructor (a parent window, a resource identifier, the control's *x*, *y*, *h*, and *w* dimensions, and an optional module pointer), and also style and maximum text length parameters. This constructor is declared like this:

```
TComboBox(TWindow *parent,
          int resourceId,
          int x, int y, int w, int h,
          uint32 style,
          uint16 textLen,
          TModule *module = 0);
```

All combo boxes have the styles WS_CHILD, WS_VISIBLE, WS_GROUP, WS_TABSTOP, CBS_SORT (to sort the list items), CBS_AUTOHSCROLL (to let the user enter more text than fits in the visible edit area), and WS_VSCROLL (vertical scroll bar). The style parameter you supply is one of the Windows combo box styles CBS_SIMPLE, CBS_DROPDOWN, or CBS_DROPDOWNLIST. The text length specifies the maximum number of characters allowed in the edit area.

The second *TComboBox* constructor lets you create an ObjectWindows object that serves as an alias for an existing combo box. This constructor looks like this:

```
TComboBox(TWindow* parent,
          int resourceId,
          UINT textLen = 0,
          TModule* module = 0);
```

The following lines show a typical combo box constructor call, constructing a drop down list combo box with an unsorted list:

```
Combo1 = new TComboBox(this, ID_COMBO1, 190, 30, 150, 100, CBS_SIMPLE, 20);
Combo1->Attr.Style &= ~CBS_SORT;
```

### Modifying combo boxes

*TComboBox* defines several member functions for modifying a combo box's list and edit areas. The following table summarizes these member functions.

Because *TComboBox* is derived from *TListBox*, you can also use *TListBox* member functions to manipulate a combo box's list area.

**Table 11.13** TComboBox member functions for modifying combo boxes

| Member function | Description |
|---|---|
| *SetText* | Replace all text in the edit area. |
| *SetEditSel* | Select text in the edit area. |
| *Clear* | Delete all text in the edit area. |
| *ShowList* or *ShowList(true)* | Show the list area. |
| *HideList* or *ShowList(false)* | Hide the list area. |
| *SetExtendedUI* | Set the extended combo box UI. |

### Querying combo boxes

*TComboBox* adds several member functions to those inherited from *TListBox* for querying the contents of a combo box's edit and list areas. The following table summarizes these member functions.

**Table 11.14** TComboBox member functions for querying combo boxes

| Member function | Description |
|---|---|
| *GetTextLen* | Get length of text in edit area. |
| *GetText* | Retrieve all text in edit area. |
| *GetEditSel* | Get indexes of selected text in edit area. |
| *GetDroppedControlRect* | Get rectangle of dropped-down list. |
| *GetDroppedState* | Determine if list area is visible. |
| *GetExtendedUI* | Determine if combo box has extended UI. |

# Setting and reading control values

To manage complex dialog boxes or windows with many child-window controls, you might create a derived class to store and retrieve the state of the dialog box or window controls. The state of a control includes the text of an edit control, the position of a scroll bar, and whether a radio button is checked.

## Using transfer buffers

As an alternative to creating a derived class, you can use a structure to represent the state of the dialog box's or window's controls. This structure is called a *transfer buffer*

because control states are transferred to the buffer from the controls and to the controls from the buffer.

For example, your application can bring up a modal dialog box and, after the user closes it, extract information from the transfer buffer about the state of each control. Then, if the user brings up the dialog box again, you can transfer the control states from the transfer buffer. In addition, you can set the initial state of each control based on the transfer buffer. You can also explicitly transfer data in either direction at any time, such as to reset the states of the controls to their previous values. A window or modeless dialog box with controls can also use the transfer mechanism to set or retrieve state information at any time.

The transfer mechanism requires the use of ObjectWindows objects to represent the controls for which you'd like to transfer data. To use the transfer mechanism, you have to do three things:

- Define the transfer buffer, with an instance variable for each control for which you want to transfer data.
- Define the corresponding window or dialog box.
- Transfer the data.

## Defining the transfer buffer

The transfer buffer is a structure with one member for each control participating in the transfer. These members are known as *instance variables*. A window or dialog box can also have controls with no states to transfer. For example, by default, buttons, group boxes, and static controls don't participate in transfer. The type of the control determines the type of member needed in the transfer buffer.

To define a transfer buffer, define an instance variable for each participating control in the dialog box or window. It isn't necessary to define an instance variable for every control, only for those controls you want to transfer values to and from. The transfer buffer stores one of each type of control, except buttons, group boxes, and static controls. For example:

```
struct TSampleTransferStruct
{
  char editCtl[sizeOfEditCtl]; // edit control
  uint16 checkBox;              // check box
  uint16 radioButton;           // radio button
  TListBoxData listBox;       // list box
  TComboBoxData comboBox;     // combo box
  TScrollBarData scrollBar;   // scroll bar
};
```

Each type of control has different information to store. The following table explains the transfer buffer for each of ObjectWindows' controls.

**Table 11.15** Transfer buffer members for each type of control

| Control type | Type | Description |
|---|---|---|
| Static | **char** *array* | A character array up to the maximum length of text allowed, plus the terminating NULL. By default, static controls don't participate in transfer, but you can explicitly enable them. |
| Edit | **char** *array* | A character array up to the maximum length of text allowed, plus the terminating NULL. |
| List box | *TListBoxData* | An instance of the *TListBoxData* class; *TListBoxData* has several members for holding the list box strings, item data, and the selected indexes. |
| Combo box | *TComboBoxData* | An instance of the *TComboBoxData* class; *TComboBoxData* has several members for holding the combo box list area strings, item data, the selection index, and the contents of the edit area. |
| Check box or radio button | *uint16* | BF_CHECKED, BF_UNCHECKED, or BF_GRAYED, indicating the selection box state. |
| Scroll bar | *TScrollBarData* | An instance of *TScrollBarData*; *TScrollBarData* has three **int** members: *LowValue* to hold the minimum range; *HighValue* to hold the maximum range; and *Position* to hold the current thumb position. |

## List box transfer

Because list boxes need to transfer several pieces of information (strings, item data, and selection indexes), the transfer buffer uses a class called *TListBoxData*. *TListBoxData* has several data members to hold the list box information:

**Table 11.16** TListBoxData data members

| Data member | Type | Description |
|---|---|---|
| ItemDatas | *TUint32Array** | Contains the item data *uint32* for each item in the list box. |
| SelIndices | *TIntArray** | Contains the indexes of each selected string (in a multiple-selection list box). |
| Strings | *TStringArray** | Contains all the strings in the list box. |

*TListBoxData* also has member functions to manipulate the list box data:

**Table 11.17** TListBoxData member functions

| Member function | Description |
|---|---|
| *AddItemData* | Adds item data to the *ItemDatas* array. |
| *AddString* | Adds a string to the *Strings* array, and optionally selects it. |
| *AddStringItem* | Adds a string to the *Strings* array, optionally selects it, and adds item data to the *ItemDatas* array. |
| *GetSelString* | Get the selected string at the given index. |
| *GetSelStringLength* | Returns the length of the selected string at the given index. |
| *ResetSelections* | Removes all selections from the *SelIndices* array. |

**Table 11.17** TListBoxData member functions (continued)

| Member function | Description |
|---|---|
| *Select* | Selects the string at the given index. |
| *SelectString* | Selects the given string. |

## Combo box transfer

Combo boxes need to transfer several pieces of information (strings, item data, selected item, and the index of the selected item). The transfer buffer for combo boxes is a class called *TComboBoxData*. *TComboBoxData* has several data members to hold the combo box information:

**Table 11.18** TComboBoxData data members

| Data member | Type | Description |
|---|---|---|
| ItemDatas | *TUint32Array** | Contains the item data *uint32* for each item in the list box. |
| Selection | **char*** | Contains the selected string. |
| Strings | *TStringArray** | Contains all the strings in the list box. |

*TComboBoxData* also has several member functions to manipulate the combo box information:

**Table 11.19** TComboBoxData member functions

| Member function | Description |
|---|---|
| *AddString* | Adds a string to the *Strings* array, and optionally selects it. |
| *AddStringItem* | Adds a string to the *Strings* array, optionally selects it, and adds item data to the *ItemDatas* array. |
| *Clear* | Clears all data. |
| *GetItemDatas* | Returns a reference to *ItemDatas*. |
| *GetSelCount* | Returns number of selected items. |
| *GetSelection* | Returns a reference to the current selection. |
| *GetSelIndex* | Returns the index of the current selection. |
| *GetSelString* | Places a copy of the current selection into a character buffer. |
| *GetSelStringLength* | Returns the length of the currently selected string. |
| *GetStrings* | Returns a reference to the entire array of strings in the combobox. |
| *ResetSelections* | Sets the current selection to a null string and sets the index to CB_ERR. |
| *Select* | Sets a string in *Strings* to be the current selection, based on an index parameter. |
| *SelectString* | Sets a string in *Strings* to be the current selection, based on matching a **const char far*** parameter. |

# Defining the corresponding window or dialog box

A window or dialog box that uses the transfer mechanism must construct its participating control objects in the exact order in which the corresponding transfer buffer members are defined. To enable transfer for a window or dialog box object, call *SetTransferBuffer* and pass a pointer to the transfer buffer.

### Using transfer with a dialog box

Because dialog boxes get their definitions and the definitions of their controls from resources, you should construct control objects using the constructors that take resource IDs. For example:

```
struct TTransferBuffer
{
  char edit[30];
  TListBoxData listBox;
  TScrollBarData scrollBar;
}
  ⋮
TTransferDialog::TTransferDialog(TWindow* parent, int resId)
  : TDialog(parent, resId),
    TWindow(parent)
{
  new TEdit(this, ID_EDIT, 30);
  new TListBox(this, ID_LISTBOX);
  new TScrollBar(this, ID_SCROLLBAR);

  SetTransferBuffer(&TTransferBuffer);
}
```

Control objects you construct like this automatically have transfer enabled (except for button, group box, and static control objects). To explicitly exclude a control from the transfer mechanism, call its *DisableTransfer* member function after constructing it.

### Using transfer with a window

Controls constructed in a window have transfer disabled by default. To enable transfer, call the control object's *EnableTransfer* member function:

```
ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
ListBox->EnableTransfer();
```

## Transferring the data

In most cases, transferring data to or from a window is automatic, but you can also explicitly transfer data at any time.

### Transferring data to a window

Transfer to a window happens automatically when you construct a window object. The constructor calls *SetupWindow* to create an interface element to represent the window object; it then calls *TransferData* to load any data from the transfer buffer. The window object's *SetupWindow* calls *SetupWindow* for each of its child windows as well, so each of the child windows has a chance to transfer its data. Because the parent window sets up its child windows in the order it constructed them, the data in the transfer buffer must appear in that same order.

## Transferring data from a dialog box

When a modal dialog box receives a command message with a control ID of IDOK, it automatically transfers data from the controls into the transfer buffer. Usually this message indicates that the user chose OK to close the dialog box, so the dialog box automatically updates its transfer buffer. Then, if you execute the dialog box again, it transfers from the transfer buffer to the controls.

## Transferring data from a window

You can explicitly transfer data in either direction at any time. For example, you might want to transfer data out of controls in a window or modeless dialog box. Or you might want to reset the state of the controls using the data in the transfer buffer in response to the user clicking a Reset or Revert button.

Use the *TransferData* member function in either case, passing the *tdSetData* enumeration to transfer from the transfer buffer to the controls or *tdGetData* to transfer from the controls to the transfer buffer. For example, you might want to call *TransferData* in the *CloseWindow* member function of a window object:

```
void
TMyWindow::CloseWindow()
{
  TransferData(tdGetData);
  TWindow::CloseWindow();
}
```

## Supporting transfer for customized controls

You might want to modify the way a particular control transfers its data, or to include a new control you define in the transfer mechanism. In either case, all you need to do is to write a *Transfer* member function for your control object. See the following table to interpret the meaning of the transfer flag parameter.

**Table 11.20** Transfer flag parameters

| Transfer flag parameter | Description |
|---|---|
| *tdGetData* | Copy data from the control to the location specified by the supplied pointer. Return the number of bytes transferred. |
| *tdSetData* | Copy the data from the transfer buffer at the supplied pointer to the control. Return the number of bytes transferred. |
| *tdSizeData* | Return the number of bytes that would be transferred. |

Chapter

12

# Gadget and gadget window objects

This chapter discusses the use of gadgets and gadget windows. In function, gadgets are similar to controls, in that they are used to gather input from or convey information to the user. But gadgets are implemented differently from controls. Unlike most other interface elements, gadgets are not windows: gadgets don't have window handles, they don't receive events and messages, and they aren't based on *TWindow*.

Instead, gadgets must be contained in a gadget window that controls the presentation of the gadget, all message processing, and so on. The gadget receives its commands and direction from the gadget window.

This chapter discusses the various kinds of gadgets implemented in ObjectWindows. It then describes the different kinds of gadget windows available for use with the gadgets.

## Gadgets

This section discusses a number of gadgets. It begins with a discussion of *TGadget*, the base class for ObjectWindows gadgets. It then discusses the other gadget classes, *TSeparatorGadget*, *TBitmapGadget*, *TControlGadget*, *TTextGadget*, and *TButtonGadget*.

### Class TGadget

All gadgets are based on the *TGadget* class. The *TGadget* class contains the basic functionality required by all gadgets, including controlling the gadget's borders and border style, setting the size of the gadget, enabling and disabling the gadget, and so on.

#### Constructing and destroying TGadget

Here is the *TGadget* constructor:

```
TGadget(int id = 0, TBorderStyle style = None);
```

where:

- *id* is an arbitrary value as the ID number for the gadget. You can use the ID to identify a particular gadget in a gadget window. Other uses for the gadget ID are discussed in the next section.
- *style* is an **enum** *TBorderStyle*. There are five possible values for *style*:
  - *None* makes the gadget with no border style; that is, it has no visible borders.
  - *Plain* makes the gadget borders visible as lines, much like the border of a window frame.
  - *Raised* makes the gadget look as if it is raised up from the gadget window.
  - *Recessed* makes the gadget look as if it is recessed into the gadget window.
  - *Embossed* makes the gadget border look as if it has an embossed ridge as a border.

The *TGadget* destructor is declared **virtual**. The only thing it does is to remove the gadget from its gadget window if that window is still valid.

## Identifying a gadget

You can identify a gadget by using the *GetId* function to access its identifier. *GetId* takes no parameters and returns an **int** that is the gadget identifier. The identifier comes from the value passed in as the first parameter of the *TGadget* constructor.

There are a number of uses for the gadget identifier:

- You can use the identifier to identify a particular gadget. If you have a large number of gadgets in a gadget window, the easiest way to determine which gadget is which is to use the gadget identifier.
- You can set the identifier to the desired event identifier when the gadget is used to generate a command. For example, a button gadget used to open a file usually has the identifier CM_FILEOPEN.
- You can set the identifier to a string identifier if you want display a text string in a message bar or status bar when the gadget is pressed. For example, suppose you have a string identifier named IDS_MYSTRING that describes your gadget. You can set the gadget identifier to IDS_MYSTRING. Then, assuming your window has a message or status bar and you've turned menu tracking on, the string IDS_MYSTRING is displayed in the message or status bar whenever you press the gadget IDS_MYSTRING.

The last two techniques are often combined. Suppose you have a command identifier CM_FILEOPEN for the File Open menu command. You can also give the gadget the identifier CM_FILEOPEN. Then when you press the gadget, the gadget window posts the CM_FILEOPEN event. Then if you have a string with the resource identifier CM_FILEOPEN, that string is displayed in the message or status bar when you press the gadget. You can see an illustration of this technique in Step 10 of the *ObjectWindows Tutorial* manual.

## Modifying and accessing gadget appearance

You can modify and check the margin width, border width, and border style of a gadget using the following functions:

```
void SetBorders(TBorders& borders);
TBorders &GetBorders();
void SetMargins(TMargins& margins);
TMargins &GetMargins();
void SetBorderStyle(TBorderStyle style);
TBorderStyle GetBorderStyle();
```

The border is the outermost boundary of a gadget. The *TBorders* structure used with the *SetBorders* and *GetBorders* functions has four data members. These **unsigned** data members, *Left*, *Right*, *Top*, and *Bottom*, contain the width of the respective borders of the gadget.

The margin is the area between the border of the gadget and the inner rectangle of the gadget. The *TMargins* structure used with the *SetMargins* and *GetMargins* functions has four data members. These **int** data members, *Left*, *Right*, *Top*, and *Bottom*, contain the width of the respective margins of the gadget.

The *TBorderStyle* **enum** used with the *SetBorderStyle* and *GetBorderStyle* functions is the same one used with the *TGadget* constructor. The various border style effects are achieved by painting the sides of the gadget borders and margins differently for each style.

## Bounding the gadget

The gadget's bounding rectangle is the entire area occupied by a gadget. It is contained in a *TRect* structure and is composed of the relative X and Y coordinates of the upper-left and lower-right corners of the gadget in the gadget window. The gadget window uses the bounding rectangle of the gadget to place the gadget. The gadget's bounding rectangle is also important in determining when the user has clicked the gadget.

To find and set the bounding rectangle of a gadget, use the following functions:

```
TRect &GetBounds();
virtual void SetBounds(TRect& rect);
```

Note that *SetBounds* is declared **virtual**. The default *SetBounds* updates only the bounding rectangle data. A derived class can override *SetBounds* to monitor changes and update the gadget's internal state.

## Shrink wrapping a gadget

You can use the *SetShrinkWrap* function to specify whether you want the gadget window to "shrink wrap" a gadget. When shrink wrapping is on for an axis, the overall size required for the gadget is calculated automatically based on the border size, margin size, and inner rectangle. This saves you from having to calculate the bounds size of the gadget manually.

You can turn shrink wrapping on and off independently for the width and height of the gadget:

```
void SetShrinkWrap(bool shrinkWrapWidth, bool shrinkWrapHeight);
```

where:

- *shrinkWrapWidth* turns horizontal shrink wrapping on or off, depending on whether *true* or *false* is passed in.
- *shrinkWrapHeight* turns vertical shrink wrapping on or off, depending on whether *true* or *false* is passed in.

## Setting gadget size

The gadget's size is the size of the bounding rectangle of the gadget. The size differs from the bounding rectangle in that it is independent of the position of the gadget. Thus, you can adjust the size of the gadget without changing the location of the gadget.

You can set the desired size of a gadget using the *SetSize* function:

```
void SetSize(TSize& size);
```

You can get use the *GetDesiredSize* function to get the size the gadget would like to be:

```
virtual void GetDesiredSize(TSize& size);
```

Even if you've set the desired size of the gadget with the *SetSize* function, you should still call the *GetDesiredSize* function to get the gadget's desired size. Gadget windows can change the desired size of a gadget during the layout process.

## Matching gadget colors to system colors

To make your interface consistent with your application user's system, you should implement the *SysColorChange* function. The gadget window calls the *SysColorChange* function of each gadget contained in the window when the window receives a WM_SYSCOLORCHANGE message, which has this syntax:

```
virtual void SysColorChange();
```

The default version of *SysColorChange* does nothing. If you want your gadgets to follow changes in system colors, you should implement this function. You should make sure to delete and reallocate any resources that are dependent on system color settings.

## TGadget public data members

There are two public data members in *TGadget*; both are **bool**s:

```
bool Clip;
bool WideAsPossible;
```

The value of *Clip* indicates whether a clipping rectangle should be applied before painting the gadget.

The value of *WideAsPossible* indicates whether the gadget should be expanded to fit the available room in the window. This is useful for such things as a text gadget in a message bar.

## Enabling and disabling a gadget

You can enable and disable a gadget using the following functions:

```
virtual void SetEnabled(bool);
bool GetEnabled();
```

Changing the state of a gadget using the default *SetEnabled* function causes the gadget's bounding rectangle to be invalidated, but not erased. A derived class can override *SetEnabled* to modify this behavior.

If your gadget generates a command, you should implement the *CommandEnable* function:

```
virtual void CommandEnable();
```

The default version of *CommandEnable* does nothing. A derived class can override this function to provide command enabling. The gadget should send a WM_COMMAND_ENABLE message to the gadget window's parent with a command-enabler object representing the gadget.

For example, here's how the *CommandEnable* function might be implemented:

```
void
TMyGadget::CommandEnable()
{
  Window->Parent->HandleMessage(WM_COMMAND_ENABLE,
                                0,
                                (LPARAM) &TMyGadgetEnabler(*Window->Parent, this));
}
```

## Deriving from TGadget

*TGadget* provides a number of **protected** access functions that you can use when deriving a gadget class from *TGadget*.

### Initializing and cleaning up

*TGadget* provides a couple **virtual** functions that give a gadget a chance to initialize or clean up:

```
virtual void Inserted();
virtual void Removed();
```

*Inserted* is called after inserting a gadget into a gadget window. *Removed* is called before removing the gadget from its gadget window. The default versions of these function do nothing.

### Painting the gadget

The *TGadget* class provides two different paint functions: *PaintBorder* and *Paint*.

The *PaintBorder* function paints the border of the gadget. This **virtual** function takes a single parameter, a *TDC* **&**, and returns **void**. *PaintBorder* implements the standard border styles. If you want to create a new border style, you need to override this function and provide the functionality for the new style. If you want to continue to provide the standard border styles, you should also call the *TGadget* version of this function. *PaintBorder* is called by the *Paint* function.

The *Paint* function is similar to the *TWindow* function *Paint*. This function takes a single parameter, a *TDC* **&**, and returns **void**. *Paint* is declared **virtual**. *TGadget*'s *PaintGadgets* function calls each gadget's *Paint* function when painting the gadget window. The default *Paint* function only calls the *PaintBorder* function. To paint the inner rectangle of the gadget's bounding rectangle, you should override this function to provide the necessary functionality.

If you're painting the gadget yourself in the *Paint* function, you often need to find the area inside the borders and margins of the gadget. This area is called the inner rectangle. You can find the inner rectangle using the *GetInnerRect* function:

```
void GetInnerRect(TRect& rect);
```

*GetInnerRect* places the coordinates of the inner rectangle into the *TRect* reference passed into it.

## Invalidating and updating the gadget

Just like a window, a gadget can be invalidated. *TGadget* provides two functions to invalidate the gadget:

```
void Invalidate(bool erase = true);
void InvalidateRect(const TRect& rect, bool erase = true);
```

These functions are similar to the *TWindow* functions *InvalidateRect* and *Invalidate*. *InvalidateRect* looks and functions much like its Windows API version, except that it omits its HWND parameters. *Invalidate* invalidates the entire bounding rectangle of the gadget. *Invalidate* takes a single parameter, a **bool** indicating whether the invalid area should be erased when it's updated. By default, this parameter is *true*. So to erase the entire area of your gadget, you need only call *Invalidate*, either specifying *true* or nothing at all for its parameter.

A related function is the *Update* function, which attempts to force an immediate update of the gadget. It is similar to the Windows API *UpdateWindow* function.

```
void Update();
```

## Mouse events in a gadget

You can track mouse events that happen inside and outside of a gadget. This happens through a number of "pseudo-event handlers" in the *TGadget* class. These functions look much like standard ObjectWindows event-handling functions, except that the names of the functions are not prefixed with *Ev*.

Gadgets don't have response tables like other ObjectWindows classes. This is because a gadget is not actually a window. All of a gadget's communication with the outside is handled through the gadget window. When a mouse event takes place in the gadget window, the window tries to determine which gadget is affected by the event. To find out if an event took place inside a particular gadget, you can call the *PtIn* function:

```
virtual bool PtIn(TPoint& point);
```

The default behavior for this function is to return *true* if *point* is within the gadget's bounding rectangle. You could override this function if you were designing an oddly shaped gadget.

When the mouse enters the bounding rectangle of a gadget, the gadget window calls the function *MouseEnter*. This function looks like this:

```
virtual void MouseEnter(uint modKeys, TPoint& point);
```

*modKeys* contains virtual key information identical to that passed-in in the standard ObjectWindows *EvMouseMove* function. This indicates whether various virtual keys are pressed. This parameter can be any combination of the following values: MK_CONTROL, MK_LBUTTON, MK_MBUTTON, MK_RBUTTON, or MK_SHIFT. See the *ObjectWindows Reference Guide* for a full explanation of these flags. *point* tells the gadget where the mouse entered the gadget.

Once the gadget window calls the gadget's *MouseEnter* function to inform the gadget that the mouse has entered the gadget's area, the gadget captures mouse movements by calling the gadget window's *GadgetSetCapture* to guarantee that the gadget's *MouseLeave* function is called.

Once the mouse leaves the gadget bounds, the gadget window calls *MouseLeave*. This function looks like this:

```
virtual void MouseLeave(uint modKeys, TPoint& point);
```

There are also a couple of functions to detect left mouse button clicks, *LButtonDown* and *LButtonUp*. The default behavior for *LButtonDown* is to capture the mouse if the **bool** flag *TrackMouse* is set. The default behavior for *LButtonUp* is to release the mouse if the **bool** flag *TrackMouse* is set. By default *TrackMouse* is not set.

```
virtual void LButtonDown(uint modKeys, TPoint& point);
virtual void LButtonUp(uint modKeys, TPoint& point);
```

When the mouse is moved inside the bounding rectangle of a gadget while mouse movements are being captured by the gadget window, the window calls the gadget's *MouseMove* function. This function looks like this:

```
virtual void MouseMove(uint modKeys, TPoint& point);
```

Like with *MouseEnter*, *modKeys* contains virtual key information. *point* tells the gadget where the mouse stopped moving.

# ObjectWindows gadget classes

ObjectWindows provides a number of classes derived from *TGadget*. These gadgets provide versatile and easy-to-use decorations and new ways to communicate with the user of your application. The gadget classes included in ObjectWindows are:

- *TSeparatorGadget*
- *TTextGadget*
- *TButtonGadget*
- *TControlGadget*
- *TBitmapGadget*

These gadgets are discussed in the following sections.

## Class TSeparatorGadget

*TSeparatorGadget* is a very simple gadget. Its only function is to take up space in a gadget window. You can use it when laying other gadgets out in a window to provide a margin of space between gadgets that would otherwise be placed border-to-border in the window.

The *TSeparatorGadget* constructor looks like this:

```
TSeparatorGadget(int size = 6);
```

The separator disables itself and turns off shrink wrapping. The *size* parameter is used for both the width and the height of the gadget. This lets you use the separator gadget for both vertical and horizontal spacing.

## Class TTextGadget

*TTextGadget* is used to display text information in a gadget window. You can specify the number of characters you want to be able to display in the gadget. You can also specify how the text should be aligned in the text gadget.

### Constructing and destroying TTextGadget

Here is the constructor for *TTextGadget*:

```
TTextGadget(int id = 0,
            TBorderStyle style = Recessed,
            TAlign alignment = Left,
            uint numChars = 10,
            const char* text = 0);
```

where:

- *id* is the gadget identifier.
- *style* is the gadget border style.
- *align* specifies how text should be aligned in the gadget. There are three possible values for the **enum** *TAlign*: *Left*, *Center*, and *Right*.
- *numChars* specifies the number of characters to be displayed in the gadget. This parameter determines the width of the gadget. The gadget calculates the required gadget width by multiplying the number of characters by the maximum character width of the current font. The height of the gadget is based on the maximum character height of the current font, plus space for the margin and border.
- *text* is a default message to be displayed in the gadget.

*~TTextGadget* automatically deletes the storage for the gadget's text string.

### Accessing the gadget text

You can get and set the text in the gadget using the *GetText* and *SetText* functions.

*GetText* takes no parameters and returns a **const char** *. You shouldn't attempt to modify the gadget text through the use of the returned pointer.

The *SetText* function takes a **const char** * and returns **void**. The gadget makes a copy of the text and stores it internally.

# Class TBitmapGadget

*TBitmapGadget* is a simple gadget that can display an array of bitmap images, one at a time. You should store the bitmaps as an array. To do this, the bitmaps should be drawn side by side in a single bitmap resource. The bitmaps should each be the same width.

## Constructing and destroying TBitmapGadget

Here is the constructor for *TBitmapGadget*:

```
TBitmapGadget(TResId bmpResId,
              int id,
              TBorderStyle style,
              int numImages,
              int startImage);
```

where:

- *bmResId* is the resource identifier for the bitmap resource.
- *id* is the gadget identifier.
- *style* is the gadget border style.
- *numImages* is the total number of images contained in the bitmap. The gadget figures the width of each single bitmap in the resource by dividing the width of the resource bitmap by *numImages*.

  For example, suppose you pass a bitmap resource to the *TBitmapGadget* constructor that is 400 pixels wide by 200 pixels high, and you specify *numImages* as 4. The constructor would divide the bitmap resource into four separate bitmaps, each one 100 pixels wide by 200 pixels high.

- *startImage* specifies which bitmap in the array should be initially displayed in the gadget.

*~TBitmapGadget* deletes the storage for the bitmap images.

## Selecting a new image

You can change the image being displayed in the gadget with the *SelectImage* function:

```
int SelectImage(int imageNum, bool immediate);
```

The *imageNum* parameter is the array index of the image you want displayed in the gadget. Specifying *true* for *immediate* causes the gadget to update the display immediately. Otherwise, the area is invalidated and updated when the next WM_PAINT message is received.

## Setting the system colors

*TBitmapGadget* implements the *SysColorChange* function so that the bitmaps track the system colors. It deletes the bitmap array, calls the *MapUIColors* function on the bitmap

resource, then re-creates the array. For more information on the *MapUIColors* function, see page 237.

## Class TButtonGadget

Button gadgets are the only type of gadget included in ObjectWindows that the user interacts with directly. Control gadgets, which are discussed in the next section, also provide a gadget that receives input from the user, but it does so through a control class. The gadget in that case only acts as an intermediary between the control and gadget window.

There are three normal button gadget states: up, down, and indeterminate. In addition the button can be highlighted when pressed in all three states.

There are two basic type of button gadgets, command gadgets and setting gadgets. Setting gadgets can be exclusive (like a radio button) or non-exclusive (like a check box). Commands can only be in the "up" state. Settings can be in all three states.

A button gadget is pressed when the left mouse button is pressed while the cursor position is inside the gadget's bounding rectangle. The gadget is highlighted when pressed.

Once the gadget has been pressed, it then captures the mouse's movements. When the mouse moves outside of the gadget's bounding rectangle without the left mouse button being released, highlighting is canceled but mouse movements are still captured by the gadget. The gadget is highlighted again when the mouse comes back into the gadget's bounding rectangle without the left mouse button being released.

When the left mouse button is released, mouse movements are no longer captured. If the cursor position is inside the bounding rectangle when the button is released, the gadget identifier is posted as a command message by the gadget window.

### Constructing and destroying TButtonGadget

Here is the *TButtonGadget* constructor:

```
TButtonGadget(TResId bmpResId,
              int id,
              TType type = Command,
              bool enabled = false,
              TState state = Up,
              bool repeat = false);
```

where:

- *bmpResId* is the resource identifier for the bitmap to be displayed in the button. The size of the bitmap determines the size of the gadget, because shrink wrapping is turned on.
- *id* is the gadget identifier. This is also the command that is posted when the gadget is pressed.
- *type* specifies the type of the gadget. The *TType* **enum** has three possible values:
  - *Command* specifies that the gadget is a command.

  - *Exclusive* specifies that the gadget is an exclusive setting button. Exclusive button gadgets that are adjacent to each other work together. You can set up exclusive groups by inserting other gadgets, such as separator gadgets or text gadgets, on either side of the group.
  - *NonExclusive* specifies that the gadget is a nonexclusive setting button.
- *enabled* specifies whether the button gadget is enabled or not when it is first created. If the corresponding command is enabled when the gadget is created, the button is automatically enabled.
- *state* is the default state of the button gadget. The **enum** *TState* can have three values: *Up*, *Down*, or *Indeterminate*.
- *repeat* indicates whether the button repeats when held down. If *repeat* is *true*, the button repeats when it is clicked and held.

The *~TButtonGadget* function deletes the bitmap resources and, if the resource information is contained in a string, deletes the storage for the string.

## Accessing button gadget information

There are a number of functions you can use to access a button gadget. These functions let you set the state of the gadget to any valid *TState* value, get the state of the button gadget, and get the button gadget type.

You can set the button gadget's state with the *SetButtonState* function:

```
void SetButtonState(TState);
```

You can find the button gadget's current state using the *GetButtonState* function:

```
TState GetButtonState();
```

You can find out what type of button a gadget is using the *GetButtonType* function:

```
TType GetButtonType();
```

## Setting button gadget style

You can modify the appearance of a button gadget using the following functions:

- You can turn corner notching on and off using the *SetNotchCorners* function:

  ```
  void SetNotchCorners(bool notchCorners=true);
  ```

- You can turn antialiasing of the button bevels on and off using the *SetAntialiasEdges* function:

  ```
  void SetAntialiasEdges(bool anti=true);
  ```

- You can change the style of the button shadow using the *SetShadowStyle* function. There are two options for the shadow style, using the **enum** *TShadowStyle*, *SingleShadow* and *DoubleShadow*:

  ```
  void SetShadowStyle(TShadowStyle style=DoubleShadow);
  ```

### Command enabling

*TButtonGadget* overrides the *TGadget* function *CommandEnable*. It is implemented to initiate a WM_COMMAND_ENABLE message for the gadget.

Here is the signature of the *TButtonGadget::CommandEnable* function:

```
void CommandEnable();
```

### Setting the system colors

*TButtonGadget* implements the *SysColorChange* function so that the gadget's bitmaps track the system colors. It rebuilds the gadget using the system colors. If the system colors have changed, these changes are reflected in the new button gadget. This is *not* set up to automatically track the system colors; that is, it is not necessarily call in response to a WM_SYSCOLORCHANGE event.

## Class TControlGadget

The *TControlGadget* is a fairly simple class that serves as an interface between a regular Windows control (such as a button, edit box, list box, and so on) and a gadget window. This lets you use a standard Windows control in a gadget window, like a control bar, status bar, and so on.

### Constructing and destroying TControlGadget

Here's the constructor for *TControlGadget*:

```
TControlGadget(TWindow& control, TBorderStyle style = None);
```

where:

- *control* is a reference to an ObjectWindows window object. This object should be a valid constructed control object.
- *style* is the gadget border style.

The *~TControlGadget* function destroys the control interface element, then deletes the storage for the control object.

# Gadget windows

Gadget windows are based on the class *TGadgetWindow*, which is derived from *TWindow*. Gadget windows are designed to hold a number of gadgets, lay them out, and display them in another window.

Gadget window provide a great deal of the functionality of the gadgets they contain. Because gadgets are not actually windows, they can't post or receive events, or directly interact with windows, or call Windows function for themselves. Anything that a gadget needs to be done must be done through the gadget window.

A gadget has little or no control over where it is laid out in the gadget window. The gadget window is responsible for placing and laying out all the gadgets it contains. Gadgets are generally laid in a line, either vertically or horizontally.

Gadget windows generally do not stand on their own, but instead are usually contained in another window. The most common parent window for a gadget window is a decorated frame window, such as *TDecoratedFrame* or *TDecoratedMDIFrame*, although the class *TToolBox* usually uses a *TFloatingFrame*.

## Constructing and destroying TGadgetWindow

Here is the constructor for *TGadgetWindow*:

```
TGadgetWindow(TWindow* parent = 0,
              TTileDirection direction = Horizontal,
              TFont* font = new TGadgetWindowFont,
              TModule* module = 0);
```

where:

- *parent* is a pointer to the parent window object.
- *direction* is an **enum** *TTileDirection*. There are two possible values for *direction*: *Horizontal* or *Vertical*.
- *font* is a pointer to a *TFont* object. This contains the font for the gadget window. By default, this is set to *TGadgetWindowFont*, which is a variable-width sans-serif font, usually Helvetica.
- *module* is passed as the *TModule* parameter for the *TWindow* base constructor. This parameter defaults to 0.

The *~TGadgetWindow* function deletes each of the gadgets contained in the gadget window. It then deletes the font object.

## Creating a gadget window

*TGadgetWindow* overrides the default *TWindow* member function *Create*. The *TGadgetWindow* version of this function chooses the initial size based on a number of criteria:

- Whether shrink wrapping was requested by any of the gadgets in the window
- The size of the gadgets contained in the window
- The direction of tiling in the gadget window
- Whether the gadget window has a border, and the size of that border

The *Create* function determines the proper size of the window based on these factors, sets the window size attributes, then calls the base *TWindow::Create* to actually create the window interface element.

## Inserting a gadget into a gadget window

For a gadget window to be useful, it needs to contain some gadgets. To place a gadget into the gadget window, use the *Insert* function:

```
virtual void Insert(TGadget& gadget,
                    TPlacement placement = After,
                    TGadget* sibling = 0);
```

where:

- *gadget* is a reference to the gadget to be inserted into the gadget window.
- *placement* indicates where the gadget should be inserted. The **enum** *TPlacement* can have two values, *Before* and *After*. If a sibling gadget is specified by the *sibling* parameter, the gadget is inserted *Before* or *After* the sibling, depending on the value of *placement*. If *sibling* is 0, the gadget is placed at the beginning of the gadgets in the window if *placement* is *Before*, and at the end of the gadgets if *placement* is *After*.
- *sibling* is a pointer to a sibling gadget.

If the gadget window has already been created, you need to call *LayoutSession* after calling *Insert*. Any gadget you insert will not appear in the window until the window has been laid out.

## Removing a gadget from a gadget window

To remove a gadget from your gadget window, use the *Remove* function:

```
virtual TGadget* Remove(TGadget& gadget);
```

where *gadget* is a reference to the gadget you want to remove from the window.

This function removes *gadget* from the gadget window. The gadget is returned as a *TGadget* *. The gadget object is not deleted. *Remove* returns 0 if the gadget is not in the window.

As with the *Insert* function, if the gadget window has already been created, you need to call *LayoutSession* after calling *Remove*. Any gadget you remove will not disappear from the window until the window has been laid out.

## Setting window margins and layout direction

You can change the margins and the layout direction either before the window is created or afterwards. To do this, use the *SetMargins* and *SetDirection* functions:

```
void SetMargins(TMargins& margins);
virtual void SetDirection(TTileDirection direction);
```

Both of these functions set the appropriate data members, then call the function *LayoutSession*, which is described in the next section.

You can find out in which direction the gadgets are laid out by calling the *GetDirection* function:

```
TTileDirection GetDirection() const;
```

## Laying out the gadgets

To lay out a gadget window, call the *LayoutSession* function.

```
virtual void LayoutSession();
```

The default behavior of the *LayoutSession* function is to check to see if the window interface element is already created. If not, the function returns without taking any further action; the window is laid out automatically when the window element is

created. But if the window element has already been created, *LayoutSession* tiles the gadgets and then invalidates the modified area of the gadget window.

A layout session is typically initiated by a change in margins, inserting or removing gadgets, or a gadget or gadget window changing size.

The actual work of tiling the gadgets is left to the function *TileGadgets*:

```
virtual TRect TileGadgets();
```

*TileGadgets* determines the space needed for each gadget and lays each gadget out in turn. It returns a *TRect* containing the area of the gadget window that was modified by laying out the gadgets.

*TileGadgets* calls the function *PositionGadget*. This lets derived classes adjust the spacing between gadgets to help in implementing a custom layout scheme.

```
virtual void PositionGadget(TGadget* previous, TGadget* next, TPoint& point);
```

This function takes the gadgets pointed to by *previous* and *next*, figures the required spacing between the gadgets, then fills in *point*. If you're tiling horizontally, then the relevant measure is contained in *point.x*. If you're tiling vertically, then the relevant measure is contained in *point.y*.

## Notifying the window when a gadget changes size

When a gadget changes size, it should call the *GadgetChangedSize* function for its gadget window. Here's the signature for this function:

```
void GadgetChangedSize(TGadget& gadget);
```

*gadget* is a reference to the gadget that changed size. The default version of this function simply initiates a layout session.

## Shrink wrapping a gadget window

You can specify whether you want the gadget window to "shrink wrap" a gadget using the *SetShrinkWrap* function. Shrink wrapping for a gadget window has a slightly different meaning than for a gadget. When a gadget window is shrink wrapped for an axis, the axis' size is calculated automatically based on the desired sizes of the gadgets laid out on that axis.

You can turn shrink wrapping on and off independently for the width and height of the gadget window:

```
void SetShrinkWrap(bool shrinkWrapWidth, bool shrinkWrapHeight);
```

where:

- *shrinkWrapWidth* turns horizontal shrink wrapping on or off, depending on whether *true* or *false* is passed in.
- *shrinkWrapHeight* turns vertical shrink wrapping on or off, depending on whether *true* or *false* is passed in.

## Accessing window font

You can find out the current font and font size using the *GetFont* and *GetFontHeight* functions:

```
TFont& GetFont();
uint GetFontHeight() const;
```

## Capturing the mouse for a gadget

A gadget is always notified when the left mouse button is pressed down within its bounding rectangle. After the button is pressed, you need to capture the mouse if you want to send notification of mouse movements. You can do this using the *GadgetSetCapture* and *GadgetReleaseCapture* functions:

```
bool GadgetSetCapture(TGadget& gadget);
void GadgetReleaseCapture(TGadget& gadget);
```

The *gadget* parameter for both functions indicates for which gadget the window should set or release the capture. The **bool** returned by *GadgetSetCapture* indicates whether the capture was successful.

These functions are usually called by a gadget in the window through the gadget's *Window* pointer to its gadget window.

## Setting the hint mode

The hint mode of a gadget dictates when hints about the gadget are displayed by the gadget window's parent. You can set the hint mode for a gadget using the *SetHintMode* function:

```
void SetHintMode(THintMode hintMode);
```

The **enum** *THintMode* has three possible values:

**Table 12.1** Hint mode flags

| hintMode | Hint displayed |
|---|---|
| NoHints | Hints are not displayed. |
| *PressHints* | Hints are displayed when the gadget is pressed until the button is released. |
| *EnterHints* | Hints are displayed when the mouse passes over the gadget; that is, when the mouse enters the gadget. |

You can find the current hint mode using the *GetHintMode* function:

```
THintMode GetHintMode();
```

Another function, the *SetHintCommand* function, determines when a hint is displayed:

```
void SetHintCommand(int id);
```

This function is usually called by a gadget through the gadget's *Window* pointer to its gadget window, but the gadget window could also call it. Essentially, *SetHintCommand* simulates a menu choice, making pressing the gadget the equivalent of selecting a menu choice.

For *SetHintCommand* to work properly with the standard ObjectWindows classes, a number of things must be in place:

- The decorated frame window parent of the gadget window must have a message or status bar.
- Hints must be on in the frame window.
- There must be a string resource with the same identifier as the gadget; that is, if the gadget identifier is CM_MYGADGET, you must also have a string resource defined as CM_MYGADGET.

### Idle action processing

Gadget windows have default idle action processing. The *IdleAction* function attempts to enable each gadget contained in the window by calling each gadget's *CommandEnable* function. The function then returns *false*.

```
bool IdleAction(long idleCount);
```

### Searching through the gadgets

Use one of the following functions to search through the gadgets contained in a gadget window:

```
TGadget* FirstGadget() const;
TGadget* NextGadget(TGadget& gadget) const;
TGadget* GadgetFromPoint(TPoint& point) const;
TGadget* GadgetWithId(int id) const;
```

- *FirstGadget* returns a pointer to the first gadget in the window's gadget list.
- *NextGadget* returns a pointer to the next gadget in the window's gadget list. If the current gadget is the last gadget in the window, *NextGadget* returns 0.
- *GadgetFromPoint* returns a pointer to the gadget that the point *point* is in. If *point* is not in a gadget, *GadgetFromPoint* returns 0.
- *GadgetWithId* returns a pointer to the gadget with the gadget identifier *id*. If no gadget in the window has that gadget identifier, *GadgetWithId* returns 0.

## Deriving from TGadgetWindow

You can derive from *TGadgetWindow* to make your own specialized gadget window. *TGadgetWindow* provides a number of **protected** access functions that you can use when deriving a gadget class from *TGadgetWindow*.

### Painting a gadget window

Just as with regular windows, *TGadgetWindow* implements the *Paint* function:

```
void Paint(TDC& dc, bool erase, TRect& rect);
```

This implementation of the *Paint* function selects the window's font into the device context and calls the function *PaintGadgets*:

```
virtual void PaintGadgets(TDC& dc, bool erase, TRect& rect);
```

*PaintGadgets* iterates through the gadgets in the window and asks each one to draw itself. Override *PaintGadgets* to implement a custom look for your window, such as separator lines, a raised look, and so on.

## Size and inner rectangle

Use the *GetDesiredSize* and *GetInnerRect* functions to find the overall desired size (that is, the size needed to accommodate the borders, margins, and the widest or highest gadget) and the size and location of the window's inner rectangle.

```
virtual void GetDesiredSize(TSize& size);
```

If shrink wrapping was requested for the window, *GetDesiredSize* calculates the size the window needs to be to accommodate the borders, margins, and the widest or highest gadget. If shrink wrapping was not requested, *GetDesiredSize* uses the current width and height. The results are then placed into *size*.

```
virtual void GetInnerRect(TRect& rect);
```

*GetInnerRect* calculates the area inside the borders and margins of the window. The results are then placed into *rect*.

You can override *GetDesiredSize* and *GetInnerRect* to leave extra room for a custom look for your window. If you override either one of these functions, you probably also need to override the other.

## Layout units

You can use three different units of measurement in a gadget window:

- Pixels, which are based on a single screen pixel
- Layout units, which are logical units defined by dividing the window font "em" into 8 vertical and 8 horizontal segments.
- Border units are based on the thickness of a window frame. This is usually equivalent to one pixel, but it could be greater at higher screen resolutions.

It is usually better to use layout units; because they are based on the font size, you don't have to worry about scaling your measures when you change window size or system metrics.

If you need to convert layout units to pixels, use the *LayoutUnitsToPixels* function:

```
int LayoutUnitsToPixels(int units);
```

where *units* is the layout unit measure you want to convert to pixels. *LayoutUnitsToPixels* returns the pixel equivalent of *units*.

You can also convert a *TMargins* object to actual pixel measurements using the *GetMargins* function:

```
void GetMargins(TMargins& margins,
                int& left,
                int& right,
                int& top,
                int& bottom);
```

where:

- *margins* is the object containing the measurements you want to convert. The measurements contained in *margins* can be in pixels, layout units, or border units.
- *left*, *right*, *top*, and *bottom* are the results of the conversion are placed.

### Message response functions

*TGadgetWindow* catches the following events:

- WM_CTLCOLOR
- WM_LBUTTONDOWN
- WM_LBUTTONUP
- WM_MOUSEMOVE
- WM_SIZE
- WM_SYSCOLORCHANGE

It also implements the corresponding event-handling functions.

# ObjectWindows gadget window classes

ObjectWindows provides a number of classes derived from *TGadgetWindow*. These windows provide a number of ways to display and lay out gadgets. The gadget window classes included in ObjectWindows are:

- *TControlBar*
- *TMessageBar*
- *TStatusBar*
- *TToolBox*

These classes are discussed in the following sections.

## Class TControlBar

The class *TControlBar* implements a control bar similar to the "tool bar" or "control bar" found along the top of the window of many popular applications. You can place any type of gadget in a control bar.

Here's the constructor for *TControlBar*:

```
TControlBar(TWindow* parent = 0,
            TTileDirection direction = Horizontal,
            TFont* font = new TGadgetWindowFont,
            TModule* module = 0);
```

where:

- *parent* is a pointer to the control bar's parent window.
- *direction* is an **enum** *TTileDirection*. There are two possible values for *direction*: *Horizontal* or *Vertical*.
- *font* is a pointer to a *TFont* object. This contains the font for the gadget window. By default, this is set to *TGadgetWindowFont*, which is a variable-width sans-serif font, usually Helvetica.
- *module* is passed as the *TModule* parameter for the *TWindow* base constructor. This parameter defaults to 0.

## Class TMessageBar

The *TMessageBar* class implements a message bar with no border and one text gadget as wide as the window. It positions itself horizontally across the bottom of its parent window.

### Constructing and destroying TMessageBar

Here's the constructor for *TMessageBar*:

```
TMessageBar(TWindow* parent = 0,
            TFont* font = new TGadgetWindowFont,
            TModule* module = 0);
```

where:

- *parent* is a pointer to the control bar's parent window.
- *font* is a pointer to a *TFont* object. This contains the font for the gadget window. By default, this is set to *TGadgetWindowFont*, which is a variable-width sans-serif font, usually Helvetica.
- *module* is passed as the *TModule* parameter for the *TWindow* base constructor. This parameter defaults to 0.

The *~TMessageBar* function deletes the object's text storage.

### Setting message bar text

Use the *SetText* function to set the text for the message bar text gadget:

```
void SetText(const char* text);
```

This function causes the string *text* to be displayed in the message bar.

### Setting the hint text

Use the *SetHintText* function to set the menu or command item hint text to be displayed in a raised field over the message bar:

```
virtual void SetHintText(const char* text);
```

If you pass *text* as 0, the hint text is cleared.

# Class TStatusBar

*TStatusBar* is similar to *TMessageBar*. The difference is that status bars have more options than a plain message bar, such as multiple text gadgets and reserved space for keyboard mode indicators such as Caps Lock, Insert or Overwrite, and so on.

## Constructing and destroying TStatusBar

Here's the constructor for *TStatusBar*:

```
TStatusBar(TWindow* parent = 0,
           TGadget::TBorderStyle borderStyle = TGadget::Recessed,
           uint modeIndicators = 0,
           TFont* font = new TGadgetWindowFont,
           TModule* module = 0);
```

where:

- *parent* is a pointer to the parent window object.
- *style* is an **enum** *TBorderStyle*.
- *modeIndicators* indicates which keyboard modes can be displayed in the status bar. A defined **enum** type called *TModeIndicator* provides the following valid values for this parameter:
  - ExtendSelection
  - CapsLock
  - NumLock
  - ScrollLock
  - Overtype
  - RecordingMacro

  These values can be ORed together to indicate multiple keyboard mode indicators.
- *font* is a pointer to a *TFont* object that contains the font for the gadget window.
- *module* is passed as the *TModule* parameter for the *TWindow* base constructor. This parameter defaults to 0.

## Inserting gadgets into a status bar

*TStatusBar* overrides the default *Insert* function. By default, the *TStatusBar* version adds the new gadget after the existing text gadgets but before the mode indicator gadgets.

You can place a gadget next to an existing gadget in the status bar by passing a pointer to the existing gadget in the *Insert* function as the new gadget's sibling. You can't insert a gadget beyond the mode indicators, however.

## Displaying mode indicators

For a particular mode indicator to appear on the status bar, you must have specified the mode when the status bar was constructed. But once the mode indicator is on the status bar, it is up to you to make any changes in the indicator. *TStatusBar* provides a number of functions to modify the mode indicators.

You can change the status of a mode indicator to any valid arbitrary state with the *SetModeIndicator* function:

```
void SetModeIndicator(TModeIndicator indicator, bool state);
```

where:

- *indicator* is the mode indicator you want to set. This can be any value from the **enum** *TModeIndicator* used in the constructor.
- *state* is the state to which you want to set the mode indicator.

You can also toggle a mode indicator with the *ToggleModeIndicator* function:

```
void ToggleModeIndicator(TModeIndicator indicator);
```

where *indicator* is the mode indicator you want to toggle. This can be any value from the **enum** *TModeIndicator* used in the constructor.

### Spacing status bar gadgets

You can vary the spacing between mode indicator gadgets on the status bar using the *SetSpacing* function:

```
void SetSpacing(TSpacing& spacing);
```

where *spacing* is a reference to a *TSpacing* object. *TSpacing* is a **struct** defined in the *TStatusBar* class. It has two data members, a *TMargins::TUnits* member named *Units* and an **int** named *Value*. The *TSpacing* constructor sets *Units* to *TMargins::LayoutUnits* and *Value* to 0.

The *TSpacing* **struct** lets you specify a unit of measurement and a number of units in a single object. When you pass this object into the *SetSpacing* command, the spacing between mode indicator gadgets is set to *Value Units*. You need to lay out the status bar before any changes take effect.

## Class TToolBox

*TToolBox* differs from the other ObjectWindows gadget window classes discussed so far in that it doesn't arrange its gadgets in a single line. Instead, it arranges them in a matrix. The columns of the matrix are all the same width (as wide as the widest gadget) and the rows of the matrix are all the same height (as high as the highest gadget). The gadgets are arranged so that the borders overlap and are hidden under the tool box's border.

*TToolBox* can be created as a client window in a *TFloatingFrame* to produce a palette-type tool box. For an example of this, see the PAINT example in the directory EXAMPLES\OWL\OWLAPPS\PAINT.

### Constructing and destroying TToolBox

Here's the constructor for *TToolBox*:

```
TToolBox(TWindow* parent,
         int numColumns = 2,
         int numRows = AS_MANY_AS_NEEDED,
```

```
TTileDirection direction = Horizontal,
TModule* module = 0);
```

where:

- *parent* is a pointer to the parent window object.
- *numColumns* is the number of columns in the tool box.
- *numRows* is the number of rows in the tool box.
- *direction* is an **enum** *TTileDirection*. There are two possible values for *direction*: *Horizontal* or *Vertical*. If *direction* is *Horizontal*, the gadgets are tiled starting at the upper left corner and moving from left to right, going down one row as each row is filled. If *direction* is *Vertical*, the gadgets are tiled starting at the upper left corner and moving down, going right one column as each column is filled.
- *module* is passed as the *TModule* parameter for the *TWindow* base constructor. This parameter defaults to 0.

You can specify the constant AS_MANY_AS_NEEDED for either *numColumns* or *numRows*, but not both. When you specify AS_MANY_AS_NEEDED for either parameter, the toolbox figures out how many divisions are needed based on the opposite dimension. For example, if you have 20 gadgets and you requested 4 columns, you would get 5 rows.

## Changing tool box dimensions

You can switch the dimensions of your tool box using the *SetDirection* function:

```
virtual void SetDirection(TTileDirection direction);
```

where *direction* is an **enum** *TTileDirection*. There are two possible values for *direction*: *Horizontal* or *Vertical*.

If *direction* is not equal to the current direction for the tool box, the tool box switches its rows and columns count. For example, suppose you have a tool box that has three columns and five rows, and is laid out vertically. If you call *SetDirection* and set *direction* to *Horizontal*, the tool box switches rows and columns, giving it five columns and three rows.

Chapter

# 13

# Printer objects

This chapter describes ObjectWindows classes that help you complete the following printing tasks:

- Creating a printer object
- Creating a printout object
- Printing window contents
- Printing a document
- Choosing and configuring a printer

Two ObjectWindows classes make these tasks easier:

- *TPrinter* encapsulates printer behavior and access to the printer drivers. It brings up a dialog box that lets the user select the desired printer and set the current settings for printing.
- *TPrintout* encapsulates the actual printout. Its relationship to the printer is similar to *TWindow's* relationship to the screen. Drawing on the screen happens in the *Paint* member function of the *TWindow* object, whereas writing to the printer happens in the *PrintPage* member function of the *TPrintout* object. To print something on the printer, the application passes an instance of *TPrintout* to an instance of *TPrinter's* *Print* member function.

## Creating a printer object

The easiest way to create a printer object is to declare a *TPrinter** within your window object that other objects in the program can use for their printing needs.

```
class MyWindow: public TFrameWindow
{
    ⋮
  protected:
    TPrinter* Printer;
    ⋮
};
```

To make the printer available, make *Printer* point to an instance of *TPrinter*. This can be done in the constructor:

```
MyWindow::MyWindow(TWindow* parent, char *title)
{
  ⋮
  Printer = new TPrinter;
}
```

You should also eliminate the printer object in the destructor:

```
MyWindow::~MyWindow()
{
  ⋮
  delete Printer;
}
```

Here's how it's done in the PRINTING.CPP example from directory OWLAPI\PRINTING:

```
class TRulerWin : public TFrameWindow
{
    ⋮
  protected:
    TPrinter* Printer;
};

TRulerWin::TRulerWin(TWindow* parent, const char* title, TModule* module)
  : TFrameWindow(parent, title, 0, false, module), TWindow(parent, title, module)
{
  ⋮
  Printer = new TPrinter;
}
```

For most applications, this is sufficient. The application's main window initializes a printer object that uses the default printer specified in WIN.INI. In some cases, however, you might have applications that use different printers from different windows simultaneously. In that case, construct a printer object in the constructors of each of the appropriate windows, then change the printer device for one or more of the printers. If the program uses different printers but not at the same time, it's probably best to use the same printer object and select different printers as needed.

Although you might be tempted to override the *TPrinter* constructor to use a printer other than the system default, the recommended procedure is to always use the default constructor, then change the device associated with the object (see page 201).

## Creating a printout object

Creating a printout object is similar to writing a *Paint* member function for a window object: you use Windows' graphics functions to generate the image you want on a device context. The window object's display context manages interactions with the screen device; the printout object's device context insulates you from the printer device in much the same way. Windows graphics functions are explained in Chapter 14.

To create a printout object,

- Derive a new object type from *TPrintout* that overrides the *PrintPage* member function. In very simple cases, that's all you need to do. See the *ObjectWindows Reference Guide* for a description of the *TPrintout* class.
- If the document has more than one page, you must also override the *HasPage* member function. It must return non-zero while there is another page to be printed. The current page number is passed as a parameter to *PrintPage.*

The printout object has fields that hold the size of the page and a device context that is already initialized to render to the printer. The printer object sets those values by calling the printout object's *SetPrintParams* member function. You should use the printout object's device context in any calls to Windows graphics functions.

Here is the class *TWindowPrintout,* derived from *TPrintout,* from the example program PRINTING.CPP:

```
class TWindowPrintout : public TPrintout
{
  public:
    TWindowPrintout(const char* title, TWindow* window);

    void GetDialogInfo(int& minPage, int& maxPage,
                       int& selFromPage, int& selToPage);
    void PrintPage(int page, TRect& rect, unsigned flags);
    void SetBanding(bool b) {Banding = b;}
    bool HasPage(int pageNumber) {return pageNumber == 1;}

  protected:
    TWindow* Window;
    bool Scale;
};
```

*GetDialogInfo* retrieves page-range information from a dialog box if page selection is possible. Since there is only one page, *GetDialogInfo* for *TWindowPrintout* looks like this:

```
void
TWindowPrintout::GetDialogInfo(int& minPage, int& maxPage,
                               int& selFromPage, int& selToPage)
{
  minPage = 0;
  maxPage = 0;
  selFromPage = selToPage = 0;
}
```

*PrintPage* must be overridden to print the contents of each page, band (if banding is enabled), or window. *PrintPage* for *TWindowPrintout* looks like this:

```
void
TWindowPrintout::PrintPage(int, TRect& rect, unsigned)
{
  // Conditionally scale the DC to the window so the printout
  // will resemble the window
  int prevMode;
  TSize oldVExt, oldWExt;
  if (Scale) {
    prevMode = DC->SetMapMode(MM_ISOTROPIC);
    TRect windowSize = Window->GetClientRect();
    DC->SetViewportExt(PageSize, &oldVExt);
    DC->SetWindowExt(windowSize.Size(), &oldWExt);
    DC->IntersectClipRect(windowSize);
    DC->DPtoLP(rect, 2);
  }

  // Call the window to paint itself
  Window->Paint(*DC, false, rect);

  // Restore changes made to the DC
  if (Scale) {
    DC->SetWindowExt(oldWExt);
    DC->SetViewportExt(oldVExt);
    DC->SetMapMode(prevMode);
  }
}
```

*SetBanding* is called with banding enabled:

```
printout.SetBanding(true);
```

*HasPage* is called after every page is printed, and by default returns *false*, which means only one page will be printed. This function must be overridden to return *true* while pages remain in multipage documents.

# Printing window contents

The simplest kind of printout to generate is a copy of a window, because windows don't have multiple pages, and window objects already know how to draw themselves on a device context.

To create a window printout object, construct a window printout object and pass it a title string and a pointer to the window you want printed:

```
TWindowPrintout printout("Ruler Test", this);
```

Often, you'll want a window to create a printout of itself in response to a menu command. Here is the message response member function that responds to the print command in PRINTING.CPP:

```
void
TRulerWin::CmFilePrint()      // Execute File:Print command
{
  if (Printer) {
    TWindowPrintout printout("Ruler Test", this);
    printout.SetBanding(true);
    Printer->Print(this, printout, true);
  }
}
```

This member function calls the printer object's *Print* member function, which passes a pointer to the parent window and a pointer to the printout object, and specifies whether or not a printer dialog box should be displayed.

*TWindowPrintout* prints itself by calling your window object's *Paint* member function (within *TWindowPrintout::PrintPage*), but with a printer device context instead of a display context.

# Printing a document

Windows sees a printout as a series of pages, so your printout object must turn a document into a series of page images for Windows to print. Just as you use window objects to paint images for Windows to display on the screen, you use printout objects to paint images on the printer.

Your printout object needs to be able to do these things:

- Set print parameters
- Calculate the total number of pages
- Draw each page on a device context
- Indicate if there are more pages

## Setting print parameters

To enable the document to paginate itself, the printer object (derived from class *TPrinter*) calls two of the printout object's member functions: *SetPrintParams* and then *GetDialogInfo*.

The *SetPrintParams* function initializes page-size and device-context variables in the printout object. It can also calculate any information needed to produce an efficient printout of individual pages. For example, *SetPrintParams* can calculate how many lines of text in the selected font can fit within the print area (using Windows API *GetTextMetrics*). If you override *SetPrintParams*, be sure to call the inherited member function, which sets the printout object's page-size and device-context defaults.

## Counting pages

After calling *SetPrintParams*, the printer object calls *GetDialogInfo*, which retrieves user page-range information from the printer dialog box. It can also be used to calculate the total number of pages based on page-size information calculated by *SetPrintParams*.

## Printing each page

After the printer object has given the document a chance to paginate itself, it calls the printout object's *PrintPage* member function for each page to be printed. The process of printing out just the part of the document that belongs on the given page is similar to deciding which portion gets drawn on a scrolling window.

When you write *PrintPage* member functions, keep these two issues in mind:

- *Device independence*. Make sure your code doesn't make assumptions about scale, aspect ratio, or colors. Those properties can vary between different video and printing devices, so you should remove any device dependencies from your code.
- *Device capabilities*. Although most video devices support all GDI operations, some printers do not. For example, many print devices, such as plotters, do not accept bitmaps at all. Others support only certain operations. When performing complex output tasks, your code should call the Windows API function *GetDeviceCaps*, which returns important information about the capabilities of a given output device.

## Indicating further pages

Printout objects have one last duty: to indicate to the printer object whether there are printable pages beyond a given page. The *HasPage* member function takes a page number as a parameter and returns a Boolean value indicating whether further pages exist. By default, *HasPage* returns *true* for the first page only. To print multiple pages, your printout object needs to override *HasPage* to return *true* if the document has more pages to print and *false* if the parameter passed is the last page.

Be sure that *HasPage* returns *false* at some point. If *HasPage* always returns *true*, printing goes into an endless loop.

## Other printout considerations

Printout objects have several other member functions you can override as needed. *BeginPrinting* and *EndPrinting* are called before and after any documents are printed, respectively. If you need special setup code, you can put it in *BeginPrinting* and undo it in *EndPrinting*.

Printing of pages takes place sequentially. That is, the printer calls *PrintPage* for each page in sequence. Before the first call to *PrintPage*, however, the printer object calls *BeginDocument*, passing the numbers of the first and last pages it prints. If your document needs to prepare to begin printing at a page other than the first, you should override *BeginDocument*. The corresponding member function, *EndDocument*, is called after the last page prints.

If multiple copies are printed, the multiple *BeginDocument* / *EndDocument* pairs can be called between *BeginPrinting* and *EndPrinting*.

## Choosing a different printer

You can associate the printer objects in your applications with any printer device installed in Windows. By default, *TPrinter* uses the Windows default printer, as specified in the [devices] section of the WIN.INI file.

There are two ways to specify an alternate printer: directly (in code) and through a user dialog box.

By far the most common way to assign a different printer is to bring up a dialog box that lets you choose from a list of installed printer devices. *TPrinter* does this automatically when you call its *Setup* member function. *Setup* displays a dialog box based on *TPrinterDialog*.

One of the buttons in the printer dialog box lets the user change the printer's configuration. The Setup button brings up a configuration dialog box defined in the printer's device driver. Your application has no control over the appearance or function of the driver's configuration dialog box.

In some cases, you might want to assign a specific printer device to your printer object, without user input. *TPrinter* has a *SetPrinter* member function that does just that. *SetPrinter* takes three strings as parameters: a device name, a driver name, and a port name.

Chapter

14

# Graphics objects

This chapter discusses the ObjectWindows encapsulation of the Windows GDI. ObjectWindows makes it easier to use GDI graphics objects and functions because it simplifies how you create and manipulate GDI objects. From simple objects such as pens and brushes to more complex objects such as fonts and bitmaps, the GDI encapsulation of the ObjectWindows library provides a simple, consistent model for graphical programming in Windows.

## GDI class organization

There are a number of ObjectWindows classes used to encapsulate GDI functionality. Most are derived from the *TGdiObject* class. *TGdiObject* provides the common functionality for all ObjectWindows GDI classes.

*TGdiObject* is the abstract base class for ObjectWindows GDI objects. It provides a base destructor, an HGDIOBJ conversion operator, and the base *GetObject* function. It also provides orphan control for true GDI objects (that is, objects derived from *TGdiObject*; other GDI objects, such as *TRegion*, *TIcon*, and *TDib*, which are derived from *TGdiBase*, are known as *pseudo-GDI objects*).

The other classes in the ObjectWindows GDI encapsulation are:

- *TDC* is the root class for encapsulating ObjectWindows GDI device contexts. You can create a *TDC* object directly or—for more specialized behavior—you can use derived classes.
- *TPen* contains the functionality of Windows pen objects. You can construct a pen object from scratch or from an existing pen handle, pen object, or logical pen (LOGPEN) structure.
- *TBrush* contains the functionality of Windows brush objects. You can construct a custom brush, creating a solid, styled, or patterned brush, or you can use an existing brush handle, brush object, or logical brush (LOGBRUSH) structure.

- *TFont* lets you easily use Windows fonts. You can construct a font with custom specifications, or from an existing font handle, font object, or logical font (LOGFONT) structure.
- *TPalette* encapsulates a GDI palette. You can construct a new palette or use existing palettes from various color table types that are used by DIBs.
- *TBitmap* contains Windows bitmaps. You can construct a bitmap from many sources, including files, bitmap handles, application resources, and more.
- *TRegion* defines a region in a window. You can construct a region in numerous shapes, including rectangles, ellipses, and polygons. *TRegion* is a pseudo-GDI object; it isn't derived from *TGdiObject*.
- *TIcon* encapsulates Windows icons. You can construct an icon from a resource or explicit information. *TIcon* is a pseudo-GDI object.
- *TCursor* encapsulates the Windows cursor. You can construct a cursor from a resource or explicit information.
- *TDib* encapsulates the device-independent bitmap (DIB) class. DIBs have no Windows handle; instead they are just a structure containing format and palette information and a collection of bits (pixels). This class provides a convenient way to work with DIBs like any other GDI object. A DIB is what is really inside a .BMP file, in bitmap resources, and what is put on the Clipboard as a DIB. *TDib* is a pseudo-GDI object.

## Changes to encapsulated GDI functions

Many of the functions in the ObjectWindows GDI classes might look familiar to you; this is because many of them have the same names and very nearly, if not exactly, the same function signature as regular Windows API functions. Because the ObjectWindows GDI classes replicate the functionality of so many Windows objects, there was no need to alter the existing terminology. Therefore, function names and signatures have been deliberately kept as close as possible to what you are used to in the standard Windows GDI functions.

Some improvements, however, have been made to the functions. These improvements, many of which are discussed in this section, include such things as cracking packed return values and using ObjectWindows objects in place of Windows-defined structures.

**Note** None of these changes are hard and fast rules; just because a function can somehow be converted doesn't mean it necessarily has been. But if you see an ObjectWindows function with the same name as a Windows API function that looks a little different, one of the following reasons should explain the change to you:

- API functions that take an object handle as a parameter often omit the handle in the ObjectWindows version. The *TGdiObject* base object maintains a handle to each object. The ObjectWindows version then uses that handle when passing the call on to Windows. For example, when selecting an object in a device context, you would normally use the *SelectObject* API function, as shown here:

```
void
SelectPen(HDC& hdc, HPEN& hpen)
{
  HPEN hpenOld;
  hpenOld = SelectObject(hdc, hpen);

  // Do something with the new pen.

  :

  // Now select the old pen again.
  SelectObject(hdc, hpenOld);
}
```

The ObjectWindows version of this function is encapsulated in the *TDC* class, which is derived from *TGdiObject*. The following example shows how the previous function would appear in a member function of a *TDC*-derived class. Notice the difference between the two calls to *SelectObject*:

```
void
SelectPen(TDC& dc, TPEN& pen)
{
  dc.SelectObject(pen);

  // Do something with the new pen.
  :

  // Now select the old pen again.
  dc.RestorePen();
}
```

- ObjectWindows GDI functions usually substitute an ObjectWindows type in place of a Windows type:
  - Windows API functions use individual parameters to specify x and y coordinate values; ObjectWindows GDI functions use *TPoint* objects.
  - Windows API functions use RECT structures to specify a rectangular area; ObjectWindows GDI functions use *TRect* objects.
  - Windows API functions use RGN structures to specify a region; ObjectWindows GDI functions use *TRegion* objects.
  - Windows API functions take HLOCAL or HGLOBAL parameters to pass an object that doesn't have a predefined Windows structure; ObjectWindows GDI functions use references to ObjectWindows objects.
- Some Windows functions return a *uint32* with data encoded in it. The *uint32* must then must be cracked to get the data from it. The ObjectWindows versions of these functions take a reference to some appropriate object as a parameter. The function then places the data into the object, relieving the programmer from the responsibility of cracking the value. These functions usually return a **bool**, indicating whether the function call was successful.

For example, the Windows version of *SetViewportOrg* returns a *uint32*, with the old value for the viewport origin contained in it. The ObjectWindows version of *SetViewportOrg* takes a *TPoint* reference in place of the two **int**s the Windows version takes as parameters. It also takes a second parameter, a *TPoint* *, in which the old viewport origins are placed.

# Working with device contexts

When working with the Windows GDI, you use a *device context* to access all devices, from windows to printers to plotters. The device context is a structure maintained by GDI that contains essential information about the device with which you are working, such as the default foreground and background colors, font, palette, and so on. ObjectWindows encapsulates device-context information in a number of device context classes, all of which are based on the *TDC* class.

*TDC* contains most of the device-context functionality you might require. The other DC-related classes are derived from *TDC* or *TDC*-derived classes. These derived classes only specialize the functionality of the *TDC* class and apply it to a discrete set of operations. Here is a description of each of the device-context classes:

- *TDC* is the root class for all GDI device contexts for ObjectWindows; it can be instantiated itself or specialized subclasses can be used to get specific behavior.
- *TWindowDC* provides access to the entire area owned by a window; this is the base for any device context class that releases its handle when done.
- *TScreenDC* provides direct access to the screen bitmap using a device context for window handle 0, which is for the whole screen with no clipping.
- *TDesktopDC* provides access to the desktop window's client area, which is the screen behind all other windows.
- *TClientDC* provides access to the client area owned by a window.
- *TPaintDC* wraps *BeginPaint* and *EndPaint* calls for use in an WM_PAINT response function.
- *TMetaFileDC* provides a device context with a metafile loaded for use.
- *TCreatedDC* lets you create a device context for a specified device.
- *TIC* lets you create an information context for a specified device.
- *TMemoryDC* provides access to a memory device context.
- *TDibDC* provides access to DIBs using the DIB.DRV driver.
- *TPrintDC* provides access to a printer device context.

## TDC class

Although the specialized device-context classes provide extra functionality tailored to each class' specific purpose, the *TDC* class provides *most* of each class' functionality. This section discusses this base functionality.

Because of the large number of functions contained in *TDC*, this section doesn't discuss every function in detail. Instead, areas of functionality contained in the *TDC* class are described, with ObjectWindows-specific functions and the most important API-like functions discussed in detail; the other functions are described in the *ObjectWindows Reference Guide*. In particular, many of the *TDC* functions look much like Windows API functions and are therefore described only briefly in this section. You can find general information on the difference between the Windows API functions and the ObjectWindows versions of those functions on page 204.

## Constructing and destroying TDC

*TDC* provides one public constructor and one public destructor. The public constructor takes an HDC, a handle to a device context. Essentially this means that you must have an existing device context before constructing a *TDC* object. Usually you don't construct a *TDC* directly, even though you can. Instead you usually use a *TDC* object when passing some device context as a function parameter or a pointer to a *TDC* to point to some device context contained in either a *TDC* or *TDC*-derived object.

*~TDC* restores all the default objects in the device context and discards the objects.

*TDC* also provides two **protected** constructors for use by derived classes. The first is a default constructor so that derived classes don't have to explicitly call *TDC*'s constructor. The second takes an HDC and a *TAutoDelete* flag. *TAutoDelete* is an **enum** that can be *NoAutoDelete* or *AutoDelete*. The *TAutoDelete* parameter is used to initialize the *ShouldDelete* member, which is inherited from *TGdiObject* (the public *TDC* constructor initializes this to *NoAutoDelete*).

## Device-context operators

*TDC* provides one conversion operator, HDC, that lets you return the handle to the device context of your particular *TDC* or *TDC*-derived object. This operator is most often invoked implicitly. When you use a *TDC* object where you would normally use an HDC, such as in a function call or the like, the compiler tries to find a way to cast the object to the required type. Thus it uses the HDC conversion operator even though it is not explicitly called.

For example, suppose you want to create a device context in memory that is compatible with the device associated with a *TDC* object. You can use the *CreateCompatibleDC* Windows API function to create the new device context from your existing *TDC* object:

```
HDC
GetCompatDC(TDC& dc, TWindow& window)
{
  HDC compatDC;

  if(!(compatDC = CreateCompatibleDC(dc))) {
    window.MessageBox("Couldn't create compatible device context!", "Failure",
                      MB_OK | MB_ICONEXCLAMATION);
    return NULL;
  } else return compatDC;
}
```

Notice that *CreateCompatibleDC* takes a single parameter, an HDC. Thus the function parameter *dc* is implicitly cast to an HDC in the *CreateCompatibleDC* call.

## Device-context functions

The functions in this section are used to access information about the device context itself. They are equivalent to the Windows API functions of the same names.

You can save and restore a device context much like normal using the functions *SaveDC* and *RestoreDC*. The following code sample shows how these functions might be used. Notice that *RestoreDC*'s single parameter uses a default value instead of specifying the **int** parameter:

```
void
TMyDC::SomeFunc(TDC& dc, int x1, int y1, int x2, int y2)
{
  dc.SaveDC();
  dc.SetMapMode(MM_LOENGLISH);
  ⋮
  dc.Rectangle(x1, -y1, x2, -y2);
  dc.RestoreDC();
}
```

You can also reset a device context to the settings contained in a DEVMODE structure using the *ResetDC* function. The only parameter *ResetDC* takes is a reference to a DEVMODE structure.

You can use the *GetDeviceCaps* function to retrieve device-specific information about a given display device. This function takes one parameter, an **int** index to the type of information to retrieve from the device context. The possible values for this parameter are the same as for the Windows API function.

You can use the *GetDCOrg* function to locate the current device context's logical coordinates within the display device's absolute physical coordinates. This function takes a reference to a *TPoint* structure and returns a **bool**. The **bool** indicates whether the function call was successful, and the *TPoint* object contains the coordinates of the device context's translation origin.

## Selecting and restoring GDI objects

You can use the *SelectObject* function to place a GDI object into a device context. There are four versions of the *SelectObject* function; all of them return **void**, but each takes different parameters. The version you should use depends on the type of object you are selecting into the device context. The different versions are:

```
SelectObject(const TBrush& brush);
SelectObject(const TPen& pen);
SelectObject(const TFont& font);
SelectObject(const TPalette& palette, bool forceBG=false);
```

In addition, *TMemoryDC* lets you select a bitmap.

Graphics objects that you can select into a device context normally exist as logical objects, which contain the information required for the creation of the object. The graphics objects are connected to the logical objects through a Windows handle. When

the graphics object is selected into the device context, a physical tool (created using the attributes contained in the logical pen) is created inside the device context.

You can also select a stock object using the function *SelectStockObject*. *SelectStockObject* takes one parameter, an **int** that is equivalent to the parameter used to call the API function *GetStockObject*. Essentially the *SelectStockObject* function takes the place of two calls: a call to *GetStockObject* to actually get a stock object, then a call to *SelectObject* to place the stock object into the device context.

*TDC* provides functions to restore original objects in a device context. There are normally four versions of this function, *RestoreBrush*, *RestorePen*, *RestoreFont*, and *RestorePalette*. A fifth, *RestoreTextBrush*, exists only for 32-bit applications. The *RestoreObjects* function calls all four functions (or five, under 32 bits), and restores all original objects in the device context. All of these functions return **void** and take no parameters.

## Drawing tool functions

*GetBrushOrg* takes one parameter, a reference to a *TPoint* object. It places the coordinates of the brush origin into the *TPoint* object. *GetBrushOrg* returns **true** if the operation was successful.

*SetBrushOrg* takes two parameters, a reference to a *TPoint* object and a *TPoint* *. This sets the device context's brush origin to the *x* and *y* values in the first *TPoint* object. If you don't specify a value for the second parameter, it defaults to 0. If you do pass a pointer to a *TPoint* object as the second parameter, *TDC::SetBrushOrg* places the old values for the brush origin into the *x* and *y* members of the object. The return value indicates whether the operation was successful.

## Color and palette functions

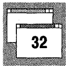
*TDC* provides a number of functions you can use to manipulate the colors and palette of a device context.

| | |
|---|---|
| GetNearestColor | RealizePalette |
| GetSystemPaletteEntries | SetSystemPaletteUse |
| GetSystemPaletteUs | UpdateColorse |

## Drawing attribute functions

Use drawing attribute functions to set the device context's drawing mode. All of these functions are analogous to the API functions of the same names, except that the HDC parameter is omitted in each.

| | |
|---|---|
| GetBkColor | SetBkColor |
| GetBkMode | SetBkMode |
| GetPolyFillMode | SetPolyFillMode |
| GetROP2 | SetROP2 |
| GetStretchBltMode | SetStretchBltMode |
| GetTextColor | SetTextColor |

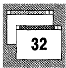


Another function, *SetMiterLimit*, is available only for 32-bit applications.

## Viewport and window mapping functions

Use these functions to set the viewport and window mapping modes:

| | |
|---|---|
| GetMapMode | GetViewportExt |
| GetViewportExt | OffsetWindowOrg |
| GetViewportOrg | ScaleViewportExt |
| GetViewportOrg | ScaleWindowExt |
| GetWindowExt | SetMapMode |
| GetWindowExt | SetViewportExt |
| GetWindowOrg | SetViewportOrg |
| GetWindowOrg | SetWindowExt |
| OffsetViewportOrg | SetWindowOrg |



The following viewport and window mapping functions are available only for 32-bit applications:

ModifyWorldTransform    SetWorldTransform

## Coordinate functions

Coordinate functions convert logical coordinates to physical coordinates and vice versa:

DPtoLP    LPtoDP

## Clip and update rectangle and region functions

Use clip and update rectangle and region functions to set up and retrieve simple or complex areas in a device context's clipping region:

| | |
|---|---|
| ExcludeClipRect | OffsetClipRgn |
| ExcludeUpdateRgn | PtVisible |
| GetBoundsRect | RectVisible |
| GetClipBox | SelectClipRgn |
| GetClipRgn | SetBoundsRect |
| IntersectClipRect | |

## Metafile functions

Use the metafile functions to access metafiles:

| | |
|---|---|
| EnumMetaFile | PlayMetaFileRecord |
| PlayMetaFile | |

## Current position functions

Use these functions to move to the current point in the device context. Three versions of *MoveTo* are provided:

- *MoveTo*(**int** *x*, **int** *y*) moves the pen to the point *x*, *y*.

- *MoveTo(TPoint &point)* moves the pen to the point *point.x, point.y*.
- *MoveTo(TPoint &point, TPoint &oldPoint)* moves the pen to the point *point.x, point.y* and places the old location of the pen into *oldPoint*.

*GetCurrentPosition* takes a reference to a *TPoint* object. It places the coordinates of the current position into the *TPoint* object and returns **true** if the function call was successful.

## Font functions

Use *TDC*'s font functions to access and manipulate fonts:

| | |
|---|---|
| EnumFontFamilies | GetCharWidth |
| EnumFonts | GetFontData |
| GetAspectRatioFilter | SetMapperFlags |
| GetCharABCWidths | |

## Path functions

Path functions are available only to 32-bit applications. The *TDC* path functions are the same as the Win32 versions, with the exception that the *TDC* versions don't take a HDC parameter.

| | |
|---|---|
| BeginPath | PathToRegion |
| CloseFigure | SelectClipPath |
| EndPath | StrokeAndFillPath |
| FillPath | StrokePath |
| FlattenPath | WidenPath |

## Output functions

*TDC* provides a great variety of output functions for all different kinds of objects that a standard device context can handle, including:

- Icons
- Rectangles
- Regions
- Shapes
- Bitmaps
- Text

Nearly all of these functions provide a number of versions: one version that provides functionality nearly identical to that of the corresponding API function (with the exception of omitting the HDC parameter) and alternate versions that use *TPoint*, *TRect*, *TRegion*, and other ObjectWindows data encapsulations to make the calls more concise and easier to understand. These functions are discussed in further detail in the *ObjectWindows Reference Guide*.

- Current position

  GetCurrentPosition MoveTo

- Icons

  DrawIcon

- Rectangles

  DrawFocusRect FillRect
  FrameRect TextRect
  InvertRect

- Regions

  FillRgn FrameRgn
  InvertRgn PaintRgn

- Shapes

  Arc Chord
  Ellipse LineDDA
  LineTo Pie
  Polygon Polyline
  PolyPolygon Rectangle
  RoundRect

- Bitmaps and blitting

  BitBlt ExtFloodFill
  FloodFill GetDIBits
  GetPixel PatBlt
  ScrollDC SetDIBits
  SetDIBitsToDevice SetPixel
  StretchBlt StretchDIBits

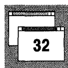
- Text

  DrawText ExtTextOut
  GrayString TabbedTextOut
  TextOut

The following functions are available for 32-bit applications only:

- Shapes

  AngleArc PolyDraw
  PolyBezier PolylineTo
  PolyBezierTo PolyPolyline




- Bitmaps and blitting

  MaskBlt PlgBlt

### Object data members and functions

These data members and functions are used to administer the device context object itself. The functions and data members discussed in this section are **protected** and can be accessed only by a *TDC*-derived class.

- *ShouldDelete* indicates whether the object should delete its handle to the device context when the destructor is invoked.
- *Handle* contains the actual handle of the device context.
- *OrgBrush*, *OrgPen*, *OrgFont*, and *OrgPalette* are the handles to the original objects when the device context was created; *OrgTextBrush* is also present in 32-bit applications.
- *CheckValid* throws an exception if the device context object is not valid.
- *Init* sets the *OrgBrush*, *OrgPen*, *OrgFont*, and *OrgPalette* when the object is created; if you're creating a *TDC*-derived class without explicitly calling a *TDC* constructor, you should call the *TDC::Init* first in your constructor.
- *GetHDC* returns an HDC using *Handle*.
- *GetAttributeHDC*, like *GetHDC*, returns an HDC using *Handle*; if you're creating an object with more than one device context, you should override this function and not *GetHDC* to provide the proper return.*OWLFastWindowFrame* draws a frame that is often used for window borders. This function uses the undocumented Windows API function *FastWindowFrame* if available, or *PatBlt* if not.

# TPen class

The *TPen* class encapsulates a logical pen. It contains a color for the pen's "ink" (encapsulated in a *TColor* object), a pen width, and the pen style.

## Constructing TPen

You can construct a *TPen* either directly, specifying the color, width, and style of the pen, or indirectly, by specifying a *TPen* & or pointer to a LOGPEN structure. Directly constructing a pen creates a new object with the specified attributes. Here is the constructor for directly constructing a pen:

```
TPen(TColor color, int width=1, int style=PS_SOLID);
```

The *style* parameter can be one of the following values: PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT, PS_NULL, or PS_INSIDEFRAME. These values are discussed in the *ObjectWindows Reference Guide*.

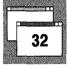
Indirectly creating a pen creates a new object, but copies the attributes of the object passed to it into the new pen object. Here are the constructors for indirectly creating a pen:

```
TPen(const LOGPEN far* logPen);
TPen(const TPen&);
```

You can also create a new *TPen* object from an existing HPEN handle:

```
TPen(HPEN handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.




Two other constructors are available only for 32-bit applications. You can use these constructors to create cosmetic or geometric pens:

```
TPen(uint32 penStyle,
     uint32 width,
     const TBrush& brush,
     uint32 styleCount,
     LPDWORD style);
TPen(uint32 penStyle,
     uint32 width,
     const LOGBRUSH& logBrush,
     uint32 styleCount,
     LPDWORD style);
```

where:

- *penStyle* is a combination of type, style, end cap, and join of the pen, where:
  - Type is either PS_GEOMETRIC or PS_COSMETIC.
  - Style can be any one of the following values:

    PS_ALTERNATE PS_DASH
    PS_DASHDOT PS_DASHDOTDOT
    PS_DOT PS_INSIDEFRAME
    PS_NULL PS_SOLID
    PS_USERSTYLE

  - End cap is specified only for geometric pens, and can be one of the following values:

    PS_ENDCAP_FLAT PS_ENDCAP_ROUND
    PS_ENDCAP_SQUARE

  - Join is specified only for geometric pens, and can be one of the following values:

    PS_JOIN_BEVEL PS_JOIN_MITER
    PS_JOIN_ROUND

- *width* is the pen width.
- *brush* or *logBrush* is a reference to an existing *TBrush* or *LOGBRUSH* object.
- *styleCount* is the size (in *uint32*s) of the *style* array; *styleCount* should be 0 unless the pen style is PS_USERSTYLE.
- *style* is a pointer to an array of *uint32*s that specifies the pattern of the pen; *style* should be NULL unless the pen style is PS_USERSTYLE.

## Accessing TPen

You can access *TPen* through an HPEN or as a LOGPEN structure. To get an HPEN from a *TPen* object, use the HPEN operator with the *TPen* object as the parameter. The HPEN operator is almost never explicitly invoked:

```
HPEN
GetHPen(TPen& pen)
{
  return pen;
}
```

This code automatically invokes the HPEN conversion operator to cast the *TPen* object to the correct type.

To convert a *TPen* object to a LOGPEN structure, use the *GetObject* function:

```
bool
GetLogPen(LOGPEN far& logPen)
{
  TPen pen(TColor::LtMagenta, 10);
  return pen.GetObject(logPen);
}
```

The following example shows how to use a pen with a *TDC* to draw a line:

```
void
TPenDemo::DrawLine(TDC& dc, const TPoint& point, TColor& color)
{
  TPen BrushPen(color, PenSize);
  dc.SelectObject(BrushPen);
  dc.LineTo(point);
}
```

# TBrush class

The *TBrush* class encapsulates a logical brush. It contains a color for the brush's ink (encapsulated in a *TColor* object), a brush width, and, depending on how the brush is constructed, the brush style, pattern, or bitmap.

## Constructing TBrush

You can construct a *TBrush* either directly, specifying the color, width, and style of the brush, or indirectly, by specifying a *TBrush* & or pointer to a LOGBRUSH structure. Directly constructing a brush creates a new object with the specified attributes. Here are the constructors for directly constructing a brush:

```
TBrush(TColor color);
TBrush(TColor color, int style);
TBrush(const TBitmap& pattern);
TBrush(const TDib& pattern);
```

The first constructor creates a solid brush with the color contained in *color*.

The second constructor creates a hatched brush with the color contained in *color* and the hatch style contained in *style*. *style* can be one of the following values:

| | |
|---|---|
| HS_BDIAGONAL | HS_CROSS |
| HS_DIAGCROSS | HS_FDIAGONAL |
| HS_HORIZONTAL | HS_VERTICAL |

The third and fourth constructors create a brush from the bitmap or DIB passed as a parameter. The width of the brush depends on the size of the bitmap or DIB.

Indirectly creating a brush creates a new object, but copies the attributes of the object passed to it into the new brush object. Here are the constructors for indirectly creating a brush:

```
TBrush(const LOGBRUSH far* logBrush);
TBrush(const TBrush& src);
```

You can also create a new *TBrush* object from an existing HBRUSH handle:

```
TBrush(HBRUSH handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.

## Accessing TBrush

You can access *TBrush* through an HBRUSH or as a LOGBRUSH structure. To get an HBRUSH from a *TBrush* object, use the HBRUSH operator with the *TBrush* object as the parameter. The HBRUSH operator is almost never explicitly invoked:

```
HBRUSH
GetHBrush(TBrush& brush)
{
  return brush;
}
```

This code automatically invokes the HBRUSH conversion operator to cast the *TBrush* object to the correct type.

To convert a *TBrush* object to a LOGBRUSH structure, use the *GetObject* function:

```
bool
GetLogBrush(LOGBRUSH far& logBrush)
{
  TBrush brush(TColor::LtCyan, HS_DIAGCROSS);
  return brush.GetObject(logBrush);
}
```

To reset the origin of a brush object, use the *UnrealizeObject* function. *UnrealizeObject* resets the brush's origin and returns nonzero if successful.

The following code shows how to use a brush to paint a rectangle in a window:

```
void
TMyWindow::PaintRect(TDC& dc, TPoint& p, TSize& size)
{
```

```
    TBrush brush(TColor(5,5,5));
    dc.SelectObject(brush);
    dc.Rectangle(p, size);
    dc.RestoreBrush();
  }
```

# TFont class

The *TFont* class lets you easily create and use Windows fonts in your applications. The *TFont* class encapsulates all attributes of a logical font.

## Constructing TFont

You can construct a *TFont* either directly, specifying all the attributes of the font in the constructor, or indirectly, by specifying a *TFont* & or pointer to a LOGFONT structure. Directly constructing a pen creates a new object with the specified attributes. Here are the constructors for directly constructing a font:

```
TFont(const char far* facename=0,
      int height=0, int width=0, int escapement=0,
      int orientation=0, int weight=FW_NORMAL,
      uint8 pitchAndFamily=DEFAULT_PITCH|FF_DONTCARE,
      uint8 italic=false, uint8 underline=false,
      uint8 strikeout=false,
      uint8 charSet=1,
      uint8 outputPrecision=OUT_DEFAULT_PRECIS,
      uint8 clipPrecision=CLIP_DEFAULT_PRECIS,
      uint8 quality=DEFAULT_QUALITY);

TFont(int height, int width, int escapement=0,
      int orientation=0,
      int weight=FW_NORMAL,
      uint8 italic=false, uint8 underline=false,
      uint8 strikeout=false,
      uint8 charSet=1,
      uint8 outputPrecision=OUT_DEFAULT_PRECIS,
      uint8 clipPrecision=CLIP_DEFAULT_PRECIS,
      uint8 quality=DEFAULT_QUALITY,
      uint8 pitchAndFamily=DEFAULT_PITCH|FF_DONTCARE,
      const char far* facename=0);
```

The first constructor lets you conveniently plug in the most commonly used attributes for a font (such as name, height, width, and so on) and let the other attributes (which generally have the same value time after time) take their default values. The second constructor has the parameters in the same order as the *CreateFont* Windows API call so you can easily cut and paste from existing Windows code.

Indirectly creating a font creates a new object, but copies the attributes of the object passed to it into the new font object. Here are the constructors for indirectly creating a font:

```
TFont(const LOGFONT far* logFont);
TFont(const TFont&);
```

You can also create a new *TFont* object from an existing HFONT handle:

```
TFont(HFONT handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.

## Accessing TFont

You can access *TFont* through an HFONT or as a LOGFONT structure. To get an HFONT from a *TFont* object, use the HFONT operator with the *TFont* object as the parameter. The HFONT operator is almost never explicitly invoked:

```
HFONT
GetHFont(TFont& font)
{
  return font;
}
```

This code automatically invokes the HFONT conversion operator to cast the *TFont* object to the correct type.

To convert a *TFont* object to a LOGFONT structure, use the *GetObject* function:

```
bool
GetLogFont(LOGFONT far& logFont)
{
  TFont font("Times Roman", 20, 8);
  return font.GetObject(logFont);
}
```

# TPalette class

The *TPalette* class encapsulates a Windows color palette that can be used with bitmaps and DIBs. *TPalette* lets you adjust the color table, match individual colors, move a palette to the Clipboard, and more.

## Constructing TPalette

You can construct a *TPalette* object either directly, passing an array of color values to the constructor, or indirectly, by specifying a *TPalette* **&**, a pointer to a LOGPALETTE structure, a pointer to a bitmap header, and so on. Directly constructing a palette creates a new object with the specified attributes. Here is the constructor for directly constructing a palette:

```
TPalette(const PALETTEENTRY far* entries, int count);
```

*entries* is an array of PALETTEENTRY objects. Each PALETTEENTRY object contains a color value specified by three separate values, one each of red, green, and blue, plus a

flags variable for the entry. *count* specifies the number of values contained in the *entries* array.

Indirectly creating a palette creates a new object, but copies the attributes of the object passed to it into the new palette object. Here are the constructors for indirectly creating a palette:

```
TPalette(const TClipboard&);
TPalette(const TPalette& palette);
TPalette(const LOGPALETTE far* logPalette);
TPalette(const BITMAPINFO far* info, uint flags=0);
TPalette(const BITMAPCOREINFO far* core, uint flags=0);
TPalette(const TDib& dib, uint flags=0);
```

Each of these constructors copies the color values contained in the object passed into the constructor into the new object. The objects passed to the constructor are not necessarily palettes themselves; many of them are objects that use palettes and contain a palette themselves. In these cases, the *TPalette* constructor extracts the palette from the object and copies it into the new palette object.

You can also create a new *TPalette* object from an existing HPALETTE handle:

```
TPalette(HPALETTE handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.

## Accessing TPalette

You can access *TPalette* through an HPALETTE or as a LOGPALETTE structure. To get an HPALETTE from a *TPalette* object, use the HPALETTE operator with the *TPalette* object as the parameter. The HPALETTE operator is almost never explicitly invoked:

```
HPALETTE
GetHPalette(TPalette& palette)
{
  return palette;
}
```

This code automatically invokes the HPALETTE conversion operator to cast the *TPalette* object to the correct type.

The *GetObject* function for *TPalette* functions the same way the Windows API call *GetObject* does when passed a handle to a palette: it places the number of entries in the color table into the *uint16* reference passed to it as a parameter. *TPalette::GetObject* returns **true** if successful.

### Member functions

*TPalette* also encapsulates a number of standard API calls for manipulating palettes:

- You can match a color with an entry in a palette using the *GetNearestPaletteIndex* function. This function takes a single parameter (a *TColor* object) and returns the index number of the closest match in the palette's color table.

- *GetNumEntries* takes no parameters and returns the number of entries in the palette's color table.
- You can get the values for a range of entries in the palette's color table using the *GetPaletteEntries* function. *TPalette::GetPaletteEntries* functions just like the Windows API call *GetPaletteEntries*, except that *TPalette::GetPaletteEntries* omits the HPALETTE parameter.
- You can set the values for a range of entries in the palette's color table using the *SetPaletteEntries* function. *TPalette::SetPaletteEntries* functions just like the Windows API call *SetPaletteEntries*, except that *TPalette::SetPaletteEntries* omits the HPALETTE parameter.
- The *GetPaletteEntry* and *SetPaletteEntry* functions work much like *GetPaletteEntries* and *SetPaletteEntries*, except that they work on a single palette entry at a time. Both functions take two parameters, the index number of a palette entry and a reference to a PALETTEENTRY object. *GetPaletteEntry* places the color value of the desired palette entry into the PALETTEENTRY object. *SetPaletteEntry* sets the palette entry indicated by the index to the value of the PALETTEENTRY object.
- You can use the *ResizePalette* function to resize a palette. *ResizePalette* takes a uint parameter, which specifies the number of entries in the resized palette. *ResizePalette* functions exactly like the Windows API *ResizePalette* call.
- The *AnimatePalette* function lets you replace entries in the palette's color table. *AnimatePalette* takes three parameters, two UINTs and a pointer to an array of PALETTEENTRY objects. The first uint specifies the first entry in the palette to be replaced. The second uint specifies the number of entries to be replaced. The entries indicated by these two UINTs are replaced by the values contained in the array of PALETTEENTRYs.
- You can also use the *UnrealizeObject* function for your palette objects. *UnrealizeObject* matches the palette to the current system palette. *UnrealizeObject* takes no parameters and functions just like the Windows API call.
- You can move a palette to the Clipboard using the *ToClipboard* function. *ToClipboard* takes a reference to a *TClipboard* object as a parameter. Because the *ToClipboard* function actually removes the object from your application, you should usually use a *TPalette* constructor to create a temporary object:

```
TClipboard clipBoard;
TPalette (tmpPalette).ToClipboard(clipBoard);
```

## Extending TPalette

*TPalette* contains two **protected**-access functions, both called *Create*. The two functions differ in that one takes BITMAPINFO * as its first parameter and the other takes a BITMAPCOREINFO * as its first parameter. These functions are called from the *TPalette* constructors that take a BITMAPINFO *, a BITMAPCOREINFO *, or a *TDib* **&**. The BITMAPINFO * and BITMAPCOREINFO * constructors call the corresponding *Create* functions. The *TDib* **&** constructor extracts a BITMAPCOREINFO * or a BITMAPINFO * from its *TDib* object and calls the appropriate *Create* function.

Both *Create* functions take a uint for their second parameter. This parameter is equivalent to the *peFlags* member of the PALETTEENTRY structure and should be passed either as a 0 or with values compatible with *peFlags*: PC_EXPLICIT, PC_NOCOLLAPSE, and PC_RESERVED. A palette entry must have the PC_RESERVED flag set to use that entry with the *AnimatePalette* function.

The *Create* functions create a LOGPALETTE using the color table from the bitmap header passed as its parameter. You can use *Create* for 2-, 16-, and 256-color bitmaps. It fails for all other types, including 24-bit DIBs. It then uses the LOGPALETTE to create the HPALETTE.

# TBitmap class

The *TBitmap* class encapsulates a Windows device-dependent bitmap, providing a number of different constructors, plus member functions to manipulate and access the bitmap.

## Constructing TBitmap

You can construct a *TBitmap* object either directly or indirectly. Using direct construction, you can specify the bitmap's width, height, and so on. Using indirect construction, you can specify an existing bitmap object, pointer to a BITMAP structure, a metafile, a *TDC* device context, and more.

Here is the constructor for directly constructing a bitmap object:

```
TBitmap(int width, int height, uint8 planes=1, uint8 count=1, void* bits=0);
```

*width* and *height* specify the width and height in pixels of the bitmap. *planes* specifies the number of color planes in the bitmap. *count* specifies the number of bits per pixel. Either *plane* or *count* must be 1. *bits* is an array containing the bits to be copied into the bitmap. *bits* can be 0, in which case the bitmap is left uninitialized.

You can create bitmap objects from existing bitmaps, either encapsulated in a *TBitmap* object or contained in a BITMAP structure.

```
TBitmap(const TBitmap& bitmap);
TBitmap(const BITMAP far* bitmap);
```

*TBitmap* provides two constructors you can use to create bitmap objects that are compatible with a given device context. The first constructor creates an uninitialized bitmap of the size *height* by *width*. Specifying **true** for the *discardable* parameter makes the bitmap discardable. A bitmap should never be discarded if it is the currently selected object in a device context.

```
TBitmap(const TDC& Dc, int width, int height, bool discardable = false);
```

The second constructor creates a bitmap compatible with the device represented by the device context from a DIB. The *usage* parameter should be CBM_INIT for 16-bit applications. CBM_INIT indicates that the bitmap should be initialized with the bits contained in the DIB object. If you don't specify CBM_INIT, the bitmap is created, but is left empty. CBM_INIT is the default.

32-bit applications can also specify CBM_CREATEDIB. The CBM_CREATEDIB flag indicates that the color format of the new bitmap should be compatible with the color format contained in the DIB's BITMAPINFO structure. If the CBM_CREATEDIB flag isn't specified, the bitmap is assumed to be compatible with the given device context.

```
TBitmap(const TDC& Dc, const TDib& dib, uint32 usage);
```

You can also create bitmaps from the Windows Clipboard, from a metafile, or from a DIB object. To create a bitmap from the Clipboard, you only need to pass a reference to a *TClipboard* object to the constructor. The constructor gets the handle of the bitmap in the Clipboard and constructs a bitmap object from the handle:

```
TBitmap(const TClipboard& clipboard);
```

To create a bitmap from a metafile, you need to pass a *TMetaFilePict* **&**, a *TPalette* **&**, and a *TSize* **&**. The constructor initializes a device-compatible bitmap (based on the palette) and plays the metafile into the bitmap:

```
TBitmap(const TMetaFilePict& metaFile, TPalette& palette, const TSize& size);
```

To create a bitmap from a device-independent bitmap, you need to pass a *TDib* **&** to the constructor. You can also specify an optional palette. The constructor creates a device context and renders the DIB into a device-compatible bitmap:

```
TBitmap(const TDib& dib, const TPalette* palette = 0);
```

You can create a bitmap object by loading it from a module. This constructor takes two parameters, first the HINSTANCE of the module containing the bitmap and second the resource ID of the bitmap you want to load:

```
TBitmap(HINSTANCE, TResId);
```

You can also create a new *TBitmap* object from an existing HBITMAP handle:

```
TBitmap(HBITMAP handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.

## Accessing TBitmap

You can access *TBitmap* through an HBITMAP or as a BITMAP structure. To get an HBITMAP from a *TBitmap* object, use the HBITMAP operator with the *TBitmap* object as the parameter. The HBITMAP operator is almost never explicitly invoked:

```
HBITMAP
GetHBitmap(TBitmap &bitmap)
{
  return bitmap;
}
```

This code automatically invokes the HBITMAP conversion operator to cast the *TBitmap* object to the correct type.

To convert a *TBitmap* object to a BITMAP structure, use the *GetObject* function:

```
bool
GetBitmap(BITMAP far& dest)
{
  TBitmap bitmap(200, 100);
  return bitmap.GetObject(dest);
}
```

The *GetObject* function fills out only the width, height, and color format information of the BITMAP structure. You can get the actual bitmap bits with the *GetBitmapBits* function.

## Member functions

*TBitmap* also encapsulates a number of standard API calls for manipulating palettes:

- You can get the same information as you get from *GetObject*, except one item at a time, using the following functions. Each function returns a characteristic of the bitmap object:

  ```
  int Width();
  int Height();
  uint8 Planes();
  uint8 BitsPixel();
  ```

- The *GetBitmapDimension* and *SetBitmapDimension* functions let you find out and change the dimensions of the bitmap. *GetBitmapDimension*, which takes a reference to a *TSize* object as its only parameter, places the size of the bitmap into the *TSize* object. *SetBitmapDimension* can take two parameters, the first a reference to a *TSize* object containing the new size for the bitmap and a pointer to a *TSize*, in which the function places the old size of the bitmap. You don't have to pass the second parameter to *SetBitmapDimension*. Both functions return **true** if the operation was successful.

  The *GetBitmapDimension* and *SetBitmapDimension* functions don't actually affect the size of the bitmap in pixels. Instead they modify only the *physical* size of the bitmap, which is often used by programs when printing or displaying bitmaps. This lets you adjust the size of the bitmap depending on the size of the physical screen.

- The *GetBitmapBits* and *SetBitmapBits* functions let you query and change the bits in a bitmap. Both functions take two parameters: a *uint32* and a **void** *. The *uint32* specifies the size of the array in bytes, and the **void** * points to an array. *GetBitmapBits* fills the array with bits from the bitmap, up to the number of bytes specified by the *uint32* parameter. *SetBitmapBits* copies the array into the bitmap, copying over the number of bytes specified in the *uint32* parameter.

- You can move a bitmap to the Clipboard using the *ToClipboard* function. *ToClipboard* takes a reference to a *TClipboard* object as a parameter. Because the *ToClipboard* function actually removes the object from your application, you should usually use a *TBitmap* constructor to create a temporary object:

```
TClipboard clipBoard;
TBitmap (tmpBitmap).ToClipboard(clipBoard);
```

## Extending TBitmap

*TBitmap* has three functions that have **protected** access: a constructor and two functions called *Create*.

The constructor is a default constructor. You can use it when constructing a derived class to prevent having to explicitly call the base class constructor. If you use the default constructor, you need to initialize the bitmap properly in your own constructor.

The first *Create* function takes a reference to a *TBitmap* object as a parameter. Essentially, this function copies the passed *TBitmap* object over to itself.

The second *Create* function takes references to a *TDib* object and to a *TPalette* object. *Create* creates a device context compatible with the *TPalette* and renders the DIB into a device-compatible bitmap.

# TRegion class

Use the *TRegion* class to define a region in a device context. You can perform a number of operations on a device context, such as painting, filling, inverting, and so on, using the region as a stencil. You can also use the *TRegion* class to define a region for your own custom operations.

## Constructing and destroying TRegion

Regions come in many shapes and sizes, from simple rectangles and rectangles with rounded corners to elaborate polygonal shapes. You can determine the shape of your region by the constructor used. You can also indirectly construct a region from a handle to a region or an existing *TRegion* object.

*TRegion* provides a default constructor that produces an empty rectangular region. You can use the function *SetRectRgn* to initialize an empty *TRegion* object. For example, suppose you derive a class from *TRegion*. In the constructor for your derived class, call *SetRectRgn* to initialize the region. This prevents you from having to call *TRegion*'s constructor explicitly:

```
class TMyRegion : public TRegion
{
  public:
    TMyRegion(TRect& rect);
    ⋮
};

TMyRegion::TMyRegion(TRect& rect)
{
  // Initialize the TRegion base with rect.
  SetRectRgn(rect);
}
```

You can directly create a *TRegion* from a number of different sources. To create a simple rectangular region, use the following constructor:

```
TRegion(const TRect& rect);
```

This creates a rectangular region from the logical coordinates in the *TRect* object.

To create a rectangular region with rounded corners, use the following constructor:

```
TRegion(const TRect& rect, const TSize& corner);
```

This creates a rectangular region from the logical coordinates in the *TRect* object, then rounds the corners into an ellipse. The height and width of the ellipse used is defined by the values in the *TSize* object.

To create an elliptical region, use the following constructor:

```
TRegion(const TRect& e, TEllipse);
```

This creates an elliptical region bounded by the logical coordinates contained in the *TRect* structure. *TEllipse* is an enumerated value with only one possible value, *Ellipse*. A call to this constructor looks something like this:

```
TRect rect(20, 20, 80, 60);
TRegion rgn(rect, TRegion::Ellipse);
```

To create regions with an irregular polygonal shape, use the following constructor:

```
TRegion(const TPoint* points, int count, int fillMode);
```

*points* is an array of *TPoint* objects. Each *TPoint* contains the logical coordinates of a vertex of the polygon. *count* indicates the number of points in the *points* array. *fillMode* indicates how the region should be filled; this can be either ALTERNATE or WINDING. There is another constructor that you can use to create regions consisting of *multiple* irregular polygonal shapes:

```
TRegion(const TPoint* points,
        const int* polyCounts,
        int count,
        int fillMode);
```

As in the other polygonal region constructor, *points* is an array of *TPoint* objects. But for this constructor, *points* contains the vertex points of a number of polygons. *polyCounts* indicates the number of points in the *points* array for each polygon. *count* indicates the total number of polygons in the region and the number of members in the *polyCount* array. *fillMode* indicates how the region should be filled; this can be either ALTERNATE or WINDING.

For example, suppose you're constructing a region that encompasses two triangular areas. Each triangle would consist of three points. Therefore *points* would have six members, three for each triangle. *polyPoints* would have two members, one for each triangle. Each member of *polyPoints* would have the value three, indicating the number of points in the *points* array that belongs to each polygon. *count* would have the value two, indicating that the region consists of two polygons.

You can create a *TRegion* from an existing HRGN:

```
TRegion(HRGN handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.

You can also create a new *TRegion* object from an existing *TRegion* object:

```
TRegion(const TRegion& region);
```

*~TRegion* deletes the region and its storage space.

## Accessing TRegion

You can access and modify *TRegion* objects directly through an HRGN handle or through a number of member functions and operators. To get an HRGN from a *TRegion* object, use the HRGN operator with the *TRegion* object as the parameter. The HRGN operator is almost never explicitly invoked:

```
HRGN
TMyBitmap::GetHRgn()
{
  return *this;
}
```

This code automatically invokes the HRGN conversion operator to cast the *TRegion* object to the correct type.

### Member functions

*TRegion* provides a number of member functions to get information from the *TRegion* object, including whether a point is contained in or touches the region:

- You can use the *SetRectRgn* function to reset the object's region to a rectangular region:

  ```
  void SetRectRgn(const TRect& rect);
  ```

  This sets the *TRegion*'s area to the logical coordinates contained in the *TRect* object passed as a parameter to the *SetRectRgn* function. The region is set to a rectangular region regardless of the shape that it previously had.

- You can use the *Contains* function to find out whether a point is contained in a region:

  ```
  bool Contains(const TPoint& point);
  ```

  *point* contains the coordinates of the point in question. *Contains* returns **true** if *point* is within the region and **false** if not.

- You can use the *Touches* function to find out whether any part of a rectangle is contained in a region:

  ```
  bool Touches(const TRect& rect);
  ```

  *rect* contains the coordinates of the rectangle in question. *Touches* returns **true** if any part of *rect* is within the region and **false** if not.

- You can use the *GetRgnBox* functions to get the coordinates of the bounding rectangle of a region:

  ```
  int GetRgnBox(TRect& box);
  TRect GetRgnBox();
  ```

The bounding rectangle is the smallest possible rectangle that encloses all of the area contained in the region. The first version of this function takes a reference to a *TRect* object as a parameter. The function places the coordinates of the bounding rectangle in the *TRect* object. The return value indicates the complexity of the region, and can be either SIMPLEREGION (region has no overlapping borders), COMPLEXREGION (region has overlapping borders), or NULLREGION (region is empty). If the function fails, the return value is ERROR.

The second version of *GetRgnBox* takes no parameters and returns a *TRect*, which contains the coordinates of the bounding rectangle. The second version of this function doesn't indicate the complexity of the region.

## Operators

*TRegion* has a large number of operators. These operators can be used to query and modify the values of a region. They aren't necessarily restricted to working with other regions; many of them let you add and subtract rectangles and other units to and from the region.

*TRegion* provides two Boolean test operators, == and !=. These operators work to compare two regions. If two regions are equivalent, the == operator returns **true**, and the != operator returns **false**. If two regions aren't equivalent, the == operator returns **false**, and the != operator returns **true**. You can use these operators much as you do their equivalents for **int**s, **char**s, and so on.

For example, suppose you want to test whether two regions are identical, and, if they're not, perform an operation on them. The code would look something like this:

```
TRegion rgn1;
TRegion rgn2;

// Initialize regions...

if(rgn1 != rgn2) {
  // Perform your operations here
  ⋮
}
```

*TRegion* also provides a number of assignment operators that you can use to change the region:

- The = operator lets you assign one region to another. For example, the statement `rgn1 = rgn2` sets the contents of *rgn1* to the contents of *rgn2*, regardless of the contexts of *rgn1* prior to the assignment.
- The += operator lets you move a region by an offset contained in a *TSize* object. This operation is analogous to numerical addition: just add the offset to each point in the region. The region retains all of its properties, except that the coordinates defining the region are shifted by the values contained in the *cx* and *cy* members of the *TSize* object:
  - If *cx* is positive, the region is shifted *cx* pixels to the right.
  - If *cx* is negative, the region is shifted *cx* pixels to the left.

- If *cy* is positive, the region is shifted *cy* pixels down.
- If *cy* is negative, the region is shifted *cy* pixels up.

For example, suppose you want to move a region to the right 50 pixels and up 20 pixels. The code would look something like this:

```
TRegion rgn;

// Initialize region...

TSize size(50, -20);
rgn += size;

// Continue working with new region.
  ⋮
```

- The –= operator, when used with a *TSize* object, does essentially the opposite of the += operator; that is, it subtracts the offset from each point in the region. For example, suppose you have the same code as in the previous example, except that instead of using the += operator, it uses the –= operator. This would offset the region in exactly the opposite way from the += operator, 50 pixels to the left and down 20 pixels.

- The –= operator, when used with a *TRegion* object, behaves differently from when it is used with a *TSize* object. To demonstrate how the –= operator works when used with *TRegion*, consider the following code:

```
TRegion rgn1, rgn2;
rgn1 -= rgn2;
```

After execution of this code, *rgn1* contains all the area it contained originally, minus any parts of that area shared by *rgn2*. Thus any point that is contained in *rgn2* is not contained in *rgn1* after this code has executed. This is analogous to subtraction: subtract the area defined by *rgn2* from *rgn1*.

- The &= operator can be used with both *TRegion* objects and *TRect* objects (before any operations are performed, the *TRect* is converted to a *TRegion* using the constructor *TRegion::TRegion*(*TRect* &)). To demonstrate how the &= operator works, consider the following code:

```
TRegion rgn1, rgn2;
rgn1 &= rgn2;
```

After execution of this code, *rgn1* contains all the area it originally shared with *rgn2*; that is, areas that were common to both regions before the execution of the &= statement. This is a logical AND operation: only the areas that are part of both *rgn1* AND *rgn2* become part of the new region.

- The |= operator can be used with both *TRegion* objects and *TRect* objects (before any operations are performed, the *TRect* is converted to a *TRegion* using the constructor *TRegion::TRegion*(*TRect* &)). To demonstrate how the |= operator works, consider the following code:

```
TRegion rgn1, rgn2;
rgn1 |= rgn2;
```

After execution of this code, *rgn1* contains all the area it originally contained, plus all the area contained in *rgn2;* that is, it contains all of both regions. This is a logical OR operation: areas that are part of either *rgn1* OR *rgn2* become part of the new region.

- The ^= operator can be used with both *TRegion* objects and *TRect* objects (before any operations are performed, the *TRect* is converted to a *TRegion* using the constructor *TRegion::TRegion(TRect* &)). To demonstrate how the ^= operator works, consider the following code:

```
TRegion rgn1, rgn2;
rgn1 ^= rgn2;
```

After execution of this code, *rgn1* contains only that area it originally contained but did *not* share with *rgn2,* plus all the area originally contained in *rgn2* that was not shared with *rgn1.* This operator combines both areas and removes the overlapping sections. This is a logical XOR (exclusive OR) operation: areas that are part of either *rgn1* OR *rgn2* but not of both become part of the new region.

# TIcon class

The *TIcon* class encapsulates an icon handle and constructors for instantiating the *TIcon* object. You can use the *TIcon* class to construct an icon from a resource or explicit info.

## Constructing TIcon

You can construct a *TIcon* in a number of ways: from an existing *TIcon* object, from a resource in the current application, from a resource in another module, or explicitly from size and data information.

You can create icon objects from an existing icon encapsulated in a *TIcon* object:

```
TIcon(HINSTANCE instance, const TIcon& icon);
```

*instance* can be any module instance. For example, you could get the instance of a DLL and get an icon from that instance:

```
TModule iconLib("MYICONS.DLL");
TIcon icon(iconLib, "MYICON");
```

Note the implicit conversion of the *TModule iconLib* into an HINSTANCE in the call to the *TIcon* constructor.

You can create a *TIcon* object from an icon resource in any module:

```
TIcon(HINSTANCE instance, TResId resId);
```

In this case, *instance* should be the HINSTANCE of the module from which you want to get the icon, and *resId* is the resource ID of the particular icon you want to get. Passing in 0 for *instance* gives you access to built-in Windows icons.

You can also load an icon from a file:

```
TIcon(HINSTANCE instance, char far* filename, int index);
```

In this case, *instance* should be the instance of the current module, *filename* is the name of the file containing the icon, and *index* is the index of the icon to be retrieved.

You can also create a new icon:

```
TIcon(HINSTANCE instance,
      TSize& size,
      int planes,
      int bitsPixel,
      void far* andBits,
      void far* xorBits);
```

In this case, *instance* should be the instance of the current module, *size* indicates the size of the icon, *planes* indicates the number of color planes, *bitsPixel* indicates the number of bits per pixel, *andBits* points to an array containing the AND mask of the icon, and *xorBits* points to an array containing the XOR mask of the icon. The *andBits* array must specify a monochrome mask. The *xorBits* array can be a monochrome or device-dependent color bitmap.

You can also create a new *TIcon* object from an existing HICON handle:

```
TIcon(HICON handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.

There are two other constructors that are available only for 32-bit applications:

```
TIcon(const void* resBits, uint32 resSize);
TIcon(const ICONINFO* iconInfo);
```

The first constructor takes two parameters: *resBits* is a pointer to a buffer containing the icon data bits (usually obtained from a call to *LookupIconIdFromDirectory* or *LoadResource* functions) and *resSize* indicates the number of bits in the *resBits* buffer.

The second constructor takes a single parameter, an ICONINFO structure. The constructor creates an icon from the information in the ICONINFO structure. The *fIcon* member of the ICONINFO structure must be **true**, indicating that the ICONINFO structure contains an icon.

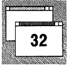
*~TIcon* deletes the icon and its storage space.

## Accessing TIcon

You can access *TIcon* through an HICON. To get an HICON from a *TIcon* object, use the HICON operator with the *TIcon* object as the parameter. The HICON operator is almost never explicitly invoked:

```
HICON
TMyIcon::GetHIcon()
{
  return *this;
}
```

This code automatically invokes the HICON conversion operator to cast the *TIcon* object to the correct type.



The other access function in *TIcon*, called *GetIconInfo*, is available for 32-bit applications only. *GetIconInfo* takes as its only parameter a pointer to a ICONINFO structure. The function fills out the ICONINFO structure and returns **true** if the operation was successful. For example, suppose you create an icon object, then want to extract the icon data into an ICONINFO structure. The code would look something like this:

```
ICONINFO iconInfo;

// Load stock icon - Exclamation
TIcon icon(0, IDI_EXCLAMATION);

icon.GetIconInfo(&iconInfo);
```

# TCursor class

The *TCursor* class encapsulates a cursor handle and constructors for instantiating the *TCursor* object. You can use the *TCursor* class to construct a cursor from a resource or explicit information.

## Constructing TCursor

You can construct a *TCursor* in a number of ways: from an existing *TCursor* object, from a resource in the current application, from a resource in another application, or explicitly from size and data information.

You can create cursor objects from an existing cursor encapsulated in a *TCursor* object:

```
TCursor(HINSTANCE instance, const TCursor& cursor);
```

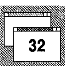
*instance* in this case should be the instance of the current application. *TCursor* does not encapsulate the application instance because *TCursor*s know nothing about application objects. It is usually easiest to access the current application instance in a window or other interface object.

```
TCursor(HINSTANCE instance, TResId resId);

TCursor(HINSTANCE instance,
        const TPoint& hotSpot,
        TSize& size,
        void far* andBits,
        void far* xorBits);
```

You can also create a new *TCursor* object from an existing HCURSOR handle:

```
TCursor(HCURSOR handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message.



There are two other constructors that are available only for 32-bit applications:

```
TCursor(const void* resBits, uint32 resSize);
TCursor(const ICONINFO* iconInfo);
```

The first constructor takes two parameters: *resBits* is a pointer to a buffer containing the cursor data bits (usually obtained from a call to *LookupIconIdFromDirectory* or *LoadResource* functions) and *resSize* indicates the number of bits in the *resBits* buffer.

The second constructor takes a single parameter, an ICONINFO structure. The constructor creates an icon from the information in the ICONINFO structure. The *fIcon* member of the ICONINFO structure must be **false**, indicating that the ICONINFO structure contains an cursor.

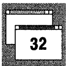
*~TCursor* deletes the cursor. If the deletion fails, the destructor throws an exception.

## Accessing TCursor

You can access *TCursor* through an HCURSOR. To get an HCURSOR from a *TCursor* object, use the HCURSOR operator with the *TCursor* object as the parameter. The HCURSOR operator is almost never explicitly invoked:

```
HCURSOR
TMyCursor::GetHCursor()
{
  return *this;
}
```

This code automatically invokes the HCURSOR conversion operator to cast the *TCursor* object to the correct type.




The other access function in *TCursor*, called *GetIconInfo*, is available for 32-bit applications only. *GetIconInfo* takes as its only parameter a pointer to a ICONINFO structure. The function fills out the ICONINFO structure and returns **true** if the operation was successful. For example, suppose you create an cursor object, then want to extract the cursor data into an ICONINFO structure. The code would look something like this:

```
ICONINFO cursorInfo;

// Load stock cursor - slashed circle
TCursor cursor(NULL, IDC_NO);

cursor.GetIconInfo(&cursorInfo);
```

# TDib class

A device-independent bitmap, or DIB, has no GDI handle like a regular bitmap, although it does have a global handle. Instead, it is just a structure containing format and palette information and a collection of bits (pixels). The *TDib* class provides a convenient way to work with DIBs like any other GDI object. The memory for the DIB is in one chunk allocated with the Windows *GlobalAlloc* functions, so that it can be passed to the Clipboard, an OLE server or client, and others outside of its instantiating application.

## Constructing and destroying TDib

You can construct a *TDib* object either directly or indirectly. Using direct construction, you can specify the bitmap's width, height, and so on. Using indirect construction, you can specify an existing bitmap object, pointer to a BITMAP structure, a metafile, a *TDC* device context, and more.

Here is the constructor for directly constructing a *TDib* object:

```
TDib(int width, int height, int nColors, uint16 mode=DIB_RGB_COLORS);
```

*width* and *height* specify the width and height in pixels of the DIB. *nColors* specifies the number of colors actually used in the DIB. *mode* can be either DIB_RGB_COLORS or DIB_PAL_COLORS. DIB_RGB_COLORS indicates that the color table consists of literal RGB values. DIB_PAL_COLORS indicates that the color table consists of an array of 16-bit indices into the currently realized logical palette.

You can create a *TDib* object by loading it from an executable application module. This constructor takes two parameters: the first is the HINSTANCE of the module containing the bitmap and the second is the resource ID of the bitmap you want to load:

```
TDib(HINSTANCE instance, TResId resId);
```

To create a *TDib* object from the Clipboard, pass a reference to a *TClipboard* object to the constructor. The constructor gets the handle of the bitmap in the Clipboard and constructs a bitmap object from the handle.

```
TDib(const TClipboard& clipboard);
```

You can load a DIB from a file (typically a .BMP file) into a *TDib* object by specifying the name as the only parameter of the constructor:

```
TDib(const char* name);
```

You can also construct a *TDib* object given a *TBitmap* object and a *TPalette* object. If no palette is give, this constructor uses the focus window's currently realized palette.

```
TDib(const TBitmap& bitmap, const TPalette* pal = 0);
```

You can create a DIB object from an existing DIB object:

```
TDib(const TDib& dib);
```

You can also create a new *TDib* object from an existing HGLOBAL handle:

```
TDib(HGLOBAL handle, TAutoDelete autoDelete = NoAutoDelete);
```

This constructor is used to obtain an ObjectWindows object as an alias to a regular Windows handle received in a message. Because an HGLOBAL handle can point to many different kinds of objects, you must ensure that the HGLOBAL you use in this constructor is actually the handle to a device-independent bitmap. If you pass a handle to another type of object, the constructor throws an exception.

If *ShouldDelete* is **true**, *~TDib* frees the resource and unlocks and frees the chunk of global memory as needed.

## Accessing TDib

*TDib* provides a number of different types of functions for accessing the encapsulated DIB.

### Type conversions

The type conversion functions for *TDib* let you access *TDib* in the most convenient manner for the operation you want to perform.

You can use the HANDLE conversion operator to access *TDib* through a HANDLE. To get a HANDLE from a *TDib* object, use the HANDLE operator with the *TDib* object as the parameter. The HANDLE operator is almost never explicitly invoked:

```
HANDLE
TMyDib::GetHandle()
{
  return *this;
}
```

This code automatically invokes the HANDLE conversion operator to cast the *TDib* object to the correct type.

You can also convert a *TDib* object to three other bitmap types. You can use the following operators to convert a *TDib* to any one of three types: BITMAPINFO *, BITMAPINFOHEADER *, or *TRgbQuad* *. You can use the result wherever that type is normally used:

```
operator BITMAPINFO far*();
operator BITMAPINFOHEADER far*();
operator TRgbQuad far*();
```

### Accessing internal structures

The functions in this section give you access to the DIB's internal data structures. These three functions return the DIB's equivalent bitmap types as pointers to BITMAPINFO, BITMAPINFOHEADER, and *TRgbQuad* objects:

```
BITMAPINFO far* GetInfo();
BITMAPINFOHEADER far* GetInfoHeader();
TRgbQuad far* GetColors();
```

The following function returns a pointer to an array of WORDs containing the color indices for the DIB:

```
uint16 far* GetIndices();
```

This function returns a pointer to an array containing the bits that make up the actual DIB image:

```
void HUGE* GetBits();
```

#### Clipboard

You can move a DIB to the Clipboard using the *ToClipboard* function. *ToClipboard* takes a reference to a *TClipboard* object as a parameter. Because the *ToClipboard* function actually

removes the object from your application, you should usually use a *TDib* constructor to create a temporary object:

```
TClipboard clipBoard;
TDib(ID_BITMAP).ToClipboard(clipBoard);
```

## DIB information

The *TDib* class provides a number of accessor functions that you can use to query a *TDib* object and get information about the DIB contained in the object:

- To find out whether the object is valid, call the *IsOK* function. The *IsOK* takes no parameters. It returns **true** if the object is valid and **false** if not.
- The *IsPM* function takes no parameters. This function returns **true** when the DIB is a Presentation Manager-compatible bitmap.
- The *Width* and *Height* functions return the bitmap's width and height respectively, in pixel units.
- The *Size* function returns the bitmap's width and height in pixel units, but contained in a *TSize* object.
- The *NumColors* function returns the number of colors used in the bitmap.
- *StartScan* is provided for compatibility with older code. This function always returns 0.
- *NumScans* is provided for compatibility with older code. This functions returns the height of the DIB in pixels.
- The *Usage* function indicates what mode the DIB is in. This value is either DIB_RGB_COLORS or DIB_PAL_COLORS.
- The *WriteFile* function writes the DIB object to disk. This function takes a single parameter, a **const char***. This should point to the name of the file in which you want to save the bitmap.

## Working in palette or RGB mode

A DIB can hold color values in two ways. In palette mode, the DIB's color table contains indices into a palette. The color values don't themselves indicate any particular color. The indices must be cross-referenced to the corresponding palette entry in the currently realized palette. In RGB mode, each entry in the DIB's color table represents an actual RGB color value.

You can switch from RGB to palette mode using these functions:

```
bool ChangeModeToPal(const TPalette& pal);
bool ChangeModeToRGB(const TPalette& pal);
```

When you switch to palette mode using *ChangeModetoPal*, the *TPalette* **&** parameter is used as the DIB's palette. Each color used in the DIB is mapped to the palette and converted to a palette index. When you switch to RGB mode using *ChangeModetoRGB*, the *TPalette* **&** parameter is used to convert the current palette indices to their RGB equivalents contained in the palette.

If you're working in RGB mode, you can use the following functions to access and modify the DIB's color table:

- Retrieve any entry in the DIB's color table using the *GetColor* function. This function takes a single parameter, an **int** indicating the index of the color table entry. *GetColor* returns a *TColor* object.
- Change any entry in the DIB's color table using the *SetColor* function. This function takes two parameters, an **int** indicating the index of the color table entry you want to change and a *TColor* containing the value to which you want to change the entry.
- Match a *TColor* object to a color table entry by using the *FindColor* function. *FindColor* takes a single parameter, a *TColor* object. *FindColor* searches through the DIB's color table until it finds an exact match for the *TColor* object. If it fails to find a match, *FindColor* returns –1.
- Substitute one color for a color that currently exists in the DIB's color table using the *MapColor* function. This function takes three parameters, a *TColor* object containing the color to be replaced, a *TColor* object containing the new color to be placed in the color table, and a **bool** that indicates whether all occurrences of the second color should be replaced. If the third parameter is **true**, all color table entries that are equal to the first parameter are replaced by the second. If the third parameter is **false**, only the first color table entry that is equal to the first parameter is replaced. By default, the third parameter is **false**. The return value of this function indicates the total number of palette entries that were replaced.

  For example, suppose you wanted to replace all occurrences of white in your DIB with light gray. The code would look something like this:

  ```
  myDib->MapColor(TColor::LtGray, TColor::White, true);
  ```

If you're working in palette mode, you can use the following functions to access and modify the DIB's color table:

- Retrieve the palette index of any color table entry using the *GetIndex* function. This function takes a single parameter, an **int** indicating the index of the color table entry. *GetIndex* returns a *uint16* containing the palette index.
- Change any entry in the DIB's color table using the *SetIndex* function. This function takes two parameters, an **int** indicating the index of the color table entry you want to change and a *uint16* containing the palette index to which you want to change the entry.
- Match a palette index to a color table entry by using the *FindIndex* function. *FindIndex* takes a single parameter, a *uint16*. *FindIndex* searches through the DIB's color table until it finds a match for the *uint16*. If it fails to find a match, *FindIndex* returns –1.
- Substitute one color for a color that currently exists in the DIB's color table using the *MapIndex* function. This function takes three parameters, a *uint16* indicating the index to be replaced, a *uint16* indicating the new palette index to be placed in the color table, and a **bool** that indicates whether all occurrences of the second color should be replaced. If the third parameter is **true**, all color table entries that are equal to the first parameter are replaced by the second. If the third parameter is **false**, only the first color table entry that is equal to the first parameter is replaced. By default, the third

parameter is **false**. The return value of this function indicates the total number of palette entries that were replaced.

## Matching interface colors to system colors

DIBs are often used to enhance and decorate a user interface. To make your interface consistent with your application user's system, you should use the *MapUIColors* function, which replaces standard interface colors with the user's own system colors. Here is the syntax for *MapUIColors*:

```
void MapUIColors(uint mapColors, TColor* bkColor = 0);
```

The *mapColors* parameter should be an OR'ed combination of five flags: *TDib::MapFace*, *TDib::MapText*, *TDib::MapShadow*, *TDib::MapHighlight*, and *TDib::MapFrame*. Each of these values causes a different color substitution to take place:

| This flag | Replaces... | With... |
|---|---|---|
| TDib::MapText | *TColor::Black* | COLOR_BTNTEXT |
| TDib::MapFace | *TColor::LtGray* | COLOR_BTNFACE |
| TDib::MapFace | *TColor::Gray* | COLOR_BTNSHADOW |
| TDib::MapFace | *TColor::White* | COLOR_BTNHIGHLIGHT |
| TDib::MapFrame | *TColor::LtMagenta* | COLOR_WINDOWFRAME |

The *bkColor* parameter, if specified, causes the color *TColor::LtYellow* to be replaced by the color *bkColor*.

Because *MapUIColors* searches for and replaces *TColor* table entries, this function is useful only with a DIB in RGB mode. Furthermore, because it replaces particular colors, you must design your interface using the standard system colors; for example, your button text should be black (*TColor::Black*), button faces should be light gray (*TColor::LtGray*), and so on. This should be fairly simple, since these are specifically designed so that they are equivalent to the standard default colors for each interface element.

You should also call the *MapUIColors* function before you modify any of the colors modified by *MapUIColors*. If you don't do this, *MapUIColors* won't be able to find the attribute color for which it is searching, and that part of the interface won't match the system colors.

# Extending TDib

*TDib* provides a number of **protected** functions that are accessible only from within *TDib* and *TDib*-derived classes. You can also access *TDib*'s control data:

- *Info* is a pointer to a BITMAPINFO or BITMAPCOREINFO structure, which contains the attributes, color table, and other information about the DIB.
- *Bits* is a **void** pointer that points to an area of memory containing the actual graphical data for the DIB.
- *NumClrs* is a **long** containing the actual number of colors used in the DIB; note that this isn't the number of colors *possible*, but the number actually used.

- *W* is an **int** indicating the width of the DIB in pixels.
- *H* is an **int** indicating the height of the DIB in pixels.
- *Mode* is a *uint16* indicating whether the DIB is in RGB mode (DIB_RGB_COLORS) or palette mode (DIB_PAL_COLORS).
- *IsCore* is a **bool**; it is **true** if the *Info* pointer points to a BITMAPCOREINFO structure and **false** if it doesn't.
- *IsResHandle* indicates whether the DIB was loaded as a resource and therefore whether *Handle* is a resource handle.

You can use the *InfoFromHandle* function to fill out the structure pointed to by *Info*. *InfoFromHandle* extracts information from *Handle* and fills out the attributes of the *Info* structure. *InfoFromHandle* takes no parameters and has no return value.

The *Read* function reads a Windows 3.0- or Presentation Manager-compatible DIB from a file referenced by a *TFile* object. When loading, *Read* checks the DIB's header, attributes, palette, and bitmap. Presentation Manager-compatible DIBs are converted to Windows DIBs on the fly. This function returns **true** if the DIB was read in correctly.

You can use the *LoadResource* function to load a DIB from an application or DLL module. This function takes two parameters, an HINSTANCE indicating the application or DLL module from which you want to load the DIB and a *TResId* indicating the particular resource within that module you want to retrieve. *LoadResource* returns **true** if the operation was successful.

You can use the *LoadFile* function to load a DIB from a file. This function takes one parameter, a **char** * that points to a string containing the name of the file containing the DIB. *LoadFile* returns **true** if the operation was successful.

Chapter

15

# Validator objects

ObjectWindows provides several ways you can associate validator objects with the edit control objects to validate the information a user types into an edit control. Using validator objects makes it easy to add data validation to existing ObjectWindows applications or to change the way a field validates its data.

This chapter discusses three topics related to data validation:

- Using the standard validator classes
- Using data validator objects
- Writing your own validator objects

At any time, you can validate the contents of any edit control by calling that object's *CanClose* member function, which in turn calls the appropriate validator object. ObjectWindows validator classes also interact at the keystroke and gain/lose focus level.

## The standard validator classes

The ObjectWindows standard validator classes automate data validation. ObjectWindows defines six validator classes in validate.h:

- *TValidator*, a base class from which all other validator classes are derived.
- *TFilterValidator*, a filter validator class.
- *TRangeValidator*, a numeric-range validator class based on *TFilterValidator*.
- *TLookupValidator*, a lookup validator base class.
- *TStringLookupValidator*, a string lookup validator class based on *TLookupValidator*.
- *TPXPictureValidator*, a picture validator class that validates a string based on a given pattern or "picture."

The following sections briefly describe each of the standard validator classes.

## Validator base class

The abstract class *TValidator* is the base class from which all validator classes are derived. *TValidator* is a validator for which all input is valid: member functions *IsValid* and *IsValidInput* always return **true**, and *Error* does nothing. Derived classes should override *IsValid*, *IsValidInput*, and *Error* to define which values are valid and when errors should be reported. Use *TValidator* as a starting point for your own validator classes if none of the other validator classes are appropriate starting points.

## Filter validator class

*TFilterValidator* is a simple validator that checks input as the user enters it. The filter validator constructor takes one parameter, a set of valid characters:

```
TFilterValidator(const TCharSet& validChars);
```

*TCharSet* is defined in bitset.h.

*TFilterValidator* overrides *IsValidInput* to return **true** only if all characters in the current input string are contained in the set of characters passed to the constructor. The edit control inserts characters only if *IsValidInput* returns **true**, so there is no need to override *IsValid*: because the characters made it through the input filter, the complete string is valid by definition. Descendants of *TFilterValidator*, such as *TRangeValidator*, can combine filtering of input with other checks on the completed string.

## Range validator class

*TRangeValidator* is a range validator derived from *TFilterValidator*. It accepts only numbers and adds range checking on the final result. The constructor takes two parameters that define the minimum and maximum valid values:

```
TRangeValidator(long min, long max);
```

The range validator constructs itself as a filter validator that accepts only the digits 0 through 9 and the plus and minus characters. The inherited *IsValidInput*, therefore, ensures that only numbers filter through. *TRangeValidator* then overrides *IsValid* to return **true** only if the entered numbers are a valid integer within the range defined in the constructor. The *Error* member function displays a message box indicating that the entered value is out of range.

## Lookup validator class

*TLookupValidator* is an abstract class that compares entered values with a list of acceptable values to determine validity. *TLookupValidator* introduces the virtual member function *Lookup*. By default, *Lookup* returns **true**. Derived classes should override *Lookup* to compare the parameter with a list of items, returning **true** if a match is found.

*TLookupValidator* overrides *IsValid* to return **true** only if *Lookup* returns **true**. In derived classes you should *not* override *IsValid*; you should instead override *Lookup*. *TStringLookupValidator* class is an instance class based on *TLookupValidator*.

## String lookup validator class

*TStringLookupValidator* is a working example of a lookup validator; it compares the string passed from the edit control with the items in a string list. If the passed-in string occurs in the list, *IsValid* returns **true**. The constructor takes only one parameter, the list of valid strings:

```
TStringLookupValidator(TSortedStringArray* strings);
```

*TSortedStringArray* is defined as

```
typedef TSArrayAsVector<string> TSortedStringArray;
```

To use a different string list after constructing the string lookup validator, use member function *NewStringList*, which disposes of the old list and installs the new list.

*TStringLookupValidator* overrides *Lookup* and *Error*. *Lookup* returns **true** if the passed-in string is in the list. *Error* displays a message box indicating that the string is not in the list.

## Picture validator class

Picture validators compare the string entered by the user with a "picture" or template that describes the format of valid input. The pictures used are compatible with those used by Borland's Paradox relational database to control user input. Constructing a picture validator requires two parameters: a string holding the template image and a Boolean value indicating whether to automatically fill-in the picture with literal characters:

```
TPXPictureValidator(const char far* pic, bool autoFill=false);
```

*TPXPictureValidator* overrides *Error*, *IsValid*, and *IsValidInput*, and adds a new member function, *Picture*. *Error* displays a message box indicating what format the string should have. *IsValid* returns **true** only if the function *Picture* returns **true**; thus you can derive new kinds of picture validators by overriding only the *Picture* member function. *IsValidInput* checks characters as the user enters them, allowing only those characters permitted by the picture format, and optionally filling in literal characters from the picture format.

Here is an example of a picture validator that is being constructed to accept social security numbers:

```
edit->SetValidator(new TPXPictureValidator("###-##-####"));
```

Picture syntax is fully described under *TPXPictureValidator* member function *Picture* in the *ObjectWindows Reference Guide*.

The *Picture* member function tries to format the given input string according to the picture format and returns a value indicating the degree of its success. The following code lists those return values:

```
// TPXPictureValidator result type
enum TPicResult
{
  prComplete,
```

```
    prIncomplete,
    prEmpty,
    prError,
    prSyntax,
    prAmbiguous,
    prIncompNoFill
  };
```

# Using data validators

To use data validator objects, you must first construct an edit control object and then construct a validator object and assign it to the edit control. From this point on, you don't need to interact with the validator object directly. The edit control knows when to call validator member functions at the appropriate times.

## Constructing an edit control object

Edit controls objects are instances of the *TEdit* class. Here is an example of how to construct an edit control:

```
TEdit* edit;
edit = new TEdit(this, 101, sizeof(transfer.NameEdit));
```

For more information on *TEdit* and using edit controls, see Chapter 11.

## Constructing and assigning validator objects

Because validator objects aren't interface objects, their constructors require only enough information to establish the validation criteria. For example, a numeric-range validator object requires only two parameters: the minimum and maximum values in the valid range.

Every edit control object has a data member that can point to a validator object. This pointer's declaration looks like this:

```
TValidator *Validator
```

If *Validator* doesn't point to a validator object, the edit control behaves as described in Chapter 11. You assign a validator by calling the edit control object's *SetValidator* member function. The edit control automatically checks with the validator object when processing key events and when called on to validate itself.

The following code shows the construction of a validator and its assignment to an edit control. In this case, a filter validator that allows only alphabetic characters is used.

```
edit->SetValidator(new TFilterValidator("A-Za-z. "));
```

A complete example showing the use of the standard validators can be found in OWLAPI\VALIDATE.

# Overriding validator member functions

Although the standard validator objects should satisfy most of your data validation needs, you can also modify the standard validators or write your own validation objects. If you decide to do this, you should be familiar with the following list of member functions inherited from the base class *TValidator*; in addition to understanding the function of each member function, you should also know how edit controls use them and how to override them if necessary.

- *Valid*
- *IsValid*
- *IsValidInput*
- *Error*

## Member function Valid

Member function *Valid* is called by the associated edit-control object to verify that the data entered is valid. Much like the *CanClose* member functions of interface objects, *Valid* is a Boolean function that returns **true** only if the string passed to it is valid data. One responsibility of an edit control's *CanClose* member function is calling the validator object's *Valid* member function, passing the edit control's current text.

When using validators with edit controls, you shouldn't need to call or override the validator's *Valid* member function; the inherited version of *Valid* will suffice. By default, *Valid* returns **true** if the member function *IsValid* returns **true**; otherwise, it calls *Error* to notify the user of the error and then returns **false**.

## Member function IsValid

The virtual member function *IsValid* is called by *Valid*, which passes *IsValid* the text string to be validated. *IsValid* returns **true** if the string represents valid data. *IsValid* does the actual data validation, so if you create your own validator objects, you'll probably override *IsValid*.

Note that you don't call *IsValid* directly. Use *Valid* to call *IsValid*, because *Valid* calls *Error* to alert the user if *IsValid* returns **false**. This separates the validation role from the error-reporting role.

## Member function IsValidInput

When an edit control object recognizes a keystroke event intended for it, it calls its validator's *IsValidInput* member function to ensure that the entered character is a valid entry. By default, *IsValidInput* member functions always return **true**, meaning that all keystrokes are acceptable, but some derived validators override *IsValidInput* to filter out unwanted keystrokes.

For example, range validators, which are used for numeric input, return **true** from *IsValidInput* only for numeric digits and the characters '+' and '–'.

*IsValidInput* takes two parameters:

```
virtual bool IsValidInput(char far* str, bool suppressFill);
```

The first parameter, *str*, points to the current input text being validated. The second parameter is a Boolean value indicating whether the validator should apply filling or padding to the input string before attempting to validate it. *TPXPictureValidator* is the only standard validator object that uses the second parameter.

## Member function Error

Virtual member function *Error* alerts the user that the contents of the edit control don't pass the validation check. The standard validator objects generally present a simple message box notifying the user that the contents of the input are invalid and describing what proper input would be.

For example, the *Error* member function for a range validator object creates a message box indicating that the value in the edit control is not between the indicated minimum and maximum values.

Although most descendant validator objects override *Error*, you should never call it directly. *Valid* calls *Error* for you if *IsValid* returns **false**, which is the only time *Error* needs to be called.

Chapter

16

# Visual Basic controls

ObjectWindows lets you use Visual Basic (VBX) 1.0-compatible controls in your Windows applications as easily as you use standard Windows or ObjectWindows controls.

VBX controls offer a wide range of functionality that is not provided in standard Windows controls. There are numerous public domain and commercial packages of VBX controls that can be used to provide a more polished and useful user interface.

This chapter describes how to design an application that uses VBX controls, describes the *TVbxControl* and *TVbxEventHandler* classes, explains how to receive messages from a VBX control, and shows how to get and set the properties of a control.

## Using VBX controls

To use VBX controls in your ObjectWindows application, follow this process:

- In your *OwlMain* function, call the function *VBXInit* before you call the *Run* function of your application object. Call the function *VBXTerm* after you call the *Run* function of your application object. *VBXInit* takes the application instance as a parameter. *VBXTerm* takes no parameters. Your *OwlMain* function might look something like this:

```
int
OwlMain(int argc, char* argv[])
{
  VBXInit(_hInstance);

  return TApplication("Wow!").Run();

  VBXTerm();
}
```

These functions initialize and close each instance's host environment necessary for using VBX controls.

- Derive a class mixing your base interface class with *TVbxEventHandler*. Your base interface class is whatever class you want to display the control in. If you're using the control in a dialog box, you need to mix in *TDialog*. The code would look something like this:

```
class MyVbxDialog : public TDialog, public TVbxEventHandler
{
  public:
    MyVbxDialog(TWindow *parent, char *name)
      : TDialog(parent, name), TWindow(parent, name) {}

  DECLARE_RESPONSE_TABLE(MyVbxDialog);
};
```

- Build a response table for the parent, including all relevant events from your control. Use the EV_VBXEVENTNAME macro to set up the response for each control event. Response tables are described in greater detail in Chapter 4.
- Create the control's parent. You can either construct the control when you create the parent or allow the parent to construct the control itself, depending on how the control is being used. This is discussed in further detail on page 248.

# VBX control classes

ObjectWindows provides two classes for use in designing an interface for VBX controls. These classes are *TVbxControl* and *TVbxEventHandler*.

## TVbxControl class

*TVbxControl* provides the actual interface to the control by letting you:

- Construct a VBX control object
- Get and change control properties
- Find the number of control properties and convert property names to and from property indices
- Find the number of control events and convert event names to and from event indices
- Call the Visual Basic 1.0 standard control methods *AddItem*, *Move*, *Refresh*, and *RemoveItem*
- Get the handle to the control element using the *TVbxControl* member function *GetHCTL*

*TVbxControl* is derived from the class *TControl*, which is derived from *TWindow*. Thus, *TVbxControl* acts much the same as any other interface element based on *TWindow*.

## TVbxControl constructors

*TVbxControl* has two constructors. The first constructor lets you dynamically construct a VBX control by specifying a VBX control file name (for example, SWITCH.VBX), control ID, control class, control title, location, and size:

```
TVbxControl(TWindow *parent,
            int id,
            const char far *FileName,
            const char far *ClassName,
            const char far *title,
            int x, int y,
            int w, int h,
            TModule *module = 0);
```

where:

- *parent* is a pointer to the control's parent.
- *id* is the control's ID, which is used when defining the parent's response table; this usually looks much like a resource ID.
- *FileName* is the name of the file that contains the VBX control, including a path name if necessary.
- *ClassName* is the class name of the control; a given VBX control file might contain a number of separate controls, each of which is identified by a unique class name (usually found in the control reference guide of third-party VBX control libraries).
- *title* is the control's title or caption.
- *x* and *y* are the coordinates within the parent object at which you want the control placed.
- *w* and *h* are the control's width and the height.
- *module* is passed to the *TControl* base constructor as the *TModule* parameter for that constructor; it defaults to 0.

The second constructor lets you set a *TVbxControl* object using a VBX control that has been defined in the application's resource file:

```
TVbxControl(TWindow *parent,
            int resId,
            TModule *module = 0);
```

where:

- *parent* is a pointer to the control's parent.
- *resId* is the resource ID of the VBX control in the resource file.
- *module* is passed to the *TControl* base constructor as the *TModule* parameter for that constructor; it defaults to 0.

## Implicit and explicit construction

You can construct VBX controls either explicitly or implicitly. You explicitly construct an object when you call one of the constructors. You implicitly construct an object when you do not call one of the constructors and allow the control to be instantiated and created by its parent.

Explicit construction involves calling either constructor of a VBX control object. This is normally done in the parent's constructor so that the VBX control is constructed and ready when the parent window is created. You can also wait to construct the control until it's needed; for example, you might want to do this if you had room for only one control. In this case, you could let the user choose a menu choice or press a button. Then, depending what the user does, you would instantiate an object and display it in an existing interface element.

The following code demonstrates explicit construction using both of the *TVbxControl* constructors in the constructor of a dialog box object:

```
class TTestDialog : public TDialog, public TVbxEventHandler
{
  public:
    TTestDialog(TWindow *parent, char *name)
      : TDialog(parent, name), TWindow(parent, name)
    {
      new TVbxControl(this, IDCONTROL1);
      new TVbxControl(this, IDCONTROL2,
                      "SWITCH.VBX", "BiSwitch",
                      "&Program VBX Control",
                      16, 70, 200, 50);
    }

  DECLARE_RESPONSE_TABLE(TTestDialog);
};
```

Implicit construction takes place when you design your interface element outside of your application source code, such as in Resource Workshop. You can use Resource Workshop to add VBX controls to dialog boxes and other interface elements. Then when you instantiate the parent object, the children, such as edit boxes, list boxes, buttons, and VBX controls, are automatically created along with the parent. The following code demonstrates how the code for this might look. It's important to note, however, that what you don't see in the following code is a VBX control. Instead, the VBX control is included in the dialog resource DIALOG_1. When DIALOG_1 is loaded and created, the VBX control is automatically created.

```
class TTestDialog : public TDialog, public TVbxEventHandler
{
  public:
    TTestDialog(TWindow *parent, char *name)
      : TDialog(parent, name), TWindow(parent, name) {}
  DECLARE_RESPONSE_TABLE(TTestDialog);
};

void
TTestWindow::CmAbout()
```

```
{
  TTestDialog(this, "DIALOG_1").Execute();
}
```

## TVbxEventHandler class

The *TVbxEventHandler* class is quite small and, for the most part, of little interest to most programmers. What it does is very important, though. Without the functionality contained in *TVbxEventHandler*, you could not communicate with your VBX controls. The event-handling programming model is described in greater detail in the following sections; this section explains only the part that *TVbxEventHandler* plays in the process.

*TVbxEventHandler* consists of a single function and a one-message response table. The function is called *EvVbxDispatch*, and it is the event-handling routine for a message called WM_VBXFIREEVENT. *EvVbxDispatch* receives the WM_VBXFIREEVENT message, converts the uncracked message to a VBXEVENT structure, and dispatches a new message, which is handled by the control's parent. Because the parent object is necessarily derived from *TVbxEventHandler*, this means that the parent calls back to itself with a different message. The new message is much easier to handle and understand. This is the message that is handled by the WM_VBXEVENTNAME macro described in the next section.

# Handling VBX control messages

You must handle VBX control messages through the control's parent object. For the parent object to be able to handle these messages, it must be derived from the class *TVbxEventHandler*. To accomplish this, you can mix whatever interface object class you want to use to contain the VBX control (for example, *TDialog*, *TFrameWindow*, or classes you might have derived from ObjectWindows interface classes) with the *TVbxEventHandler* class.

## Event response table

Once you've derived your new class, you need to build a response table for it. The response table for this class looks like a normal response table; you still need to handle all the regular command messages and events you normally do. The only addition is the EV_VBXEVENTNAME macro to handle the new class of messages from your VBX controls.

The EV_VBXEVENTNAME macro takes three parameters:

```
EV_VBXEVENTNAME(ID, Event, EvHandler)
```

where:

- *ID* is the control ID. You can find this ID either as the second parameter to both constructors or as the resource ID in the resource file.
- *Event* is a string identifying the event name. This is dependent on the control and can be one of the standard VBX event names or a custom event name. You can find this

event name by looking in the control reference guide if the control is from a third-party VBX control library.

- *EvHandler* is the handler function for this event and control. The *EvHandler* function has the signature:

```
void EvHandler(VBXEVENT FAR *event);
```

When a message is received from a VBX control by its parent, it dispatches the message to the handler function that corresponds to the correct control and event. When it calls the function, it passes it a pointer to a VBXEVENT structure. This structure is discussed in more detail in the next section.

## Interpreting a control event

Once a VBX control event has taken place and the event-handling function has been called, the function needs to deal with the VBXEVENT structure received as a parameter. This structure looks like this:

```
struct VBXEVENT
{
  HCTL hCtl;
  HWND hWnd;
  int nID;
  int iEvent;
  LPCSTR lpszEvent;
  int cParams;
  LPVOID lpParams;
};
```

where:

- *hCtl* is the handle of the sending VBX control (not a window handle).
- *hWnd* is the handle of the control window.
- *nID* is the ID of the VBX control.
- *iEvent* is the event index.
- *lpszEvent* is the event name.
- *cParams* is the number of parameters for this event.
- *lpParams* is a pointer to an array containing pointers to the parameter values for this event.

To understand this structure, you need to understand how a VBX control event works. The first three members are straightforward: they let you identify the sending control. The next two members are also fairly simple; each event that a VBX control can send has both an event index, represented here by *iEvent*, and an event name, represented here by *lpszEvent*.

The next two members, which store the parameters passed with the event, are more complex. *cParams* contains the total number of parameters available for this event. *lpParams* is an array of pointers to the event's parameters (like any other array, *lpParam* is indexed from 0 to *cParams* –1). These two members are more complicated than the previous members because there is no inherent indication of the type or meaning of each parameter. If the control is from a third-party VBX control library, you can look in

the control reference guide to find this information. Otherwise, you'll need to get the information from the designer of the control (or to have designed the control yourself).

## Finding event information

The standard way to interpret the information returned by an event is to refer to the documentation for the VBX control. Failing that, *TVbxControl* provides a number of methods for obtaining information about an event.

You can find the total number of events that a control can send by using the *TVbxControl* member function *GetNumEvents*. This returns an **int** that gives the total number of events. These events are indexed from 0 to the return value of *GetNumEvents* –1.

You can find the name of any event in this range by calling the *TVbxControl* member function *GetEventName*. *GetEventName* takes one parameter, an **int** index number, and returns a string containing the name of the event.

Conversely, you can find the index of an event by calling the *TVbxControl* member function *GetEventIndex*. *GetEventIndex* takes one parameter, a string containing the event name, and returns the corresponding **int** event index.

# Accessing a VBX control

There are two ways you can directly access a VBX control. The first way is to get and set the properties of the control. A control has a fixed number of properties you can set to affect the look or behavior of the control. The other way is to call the control's methods. A control's methods are similar to member functions in a class and are actually accessed through member functions in the *TVbxControl* class. You can use these methods to call into the object and cause an action to take place.

## VBX control properties

Every VBX control has a number of properties. Control properties affect the look and behavior of the control; for example, the colors used in various parts of the control, the size and location of the control, the control's caption, and so on. Changing these properties is usually your main way to manipulate a VBX control.

Each control's properties should be fully documented in the control reference guide of third-party VBX control libraries. If the control is not a third-party control or part of a commercial control package, then you need to consult the control's designer for any limits or special meanings to the control's properties. Many properties often function only as an index to a property. An example of this might be background patterns: 0 could mean plain, 1 could mean cross-hatched, 2 could mean black, and so on. Without the proper documentation or information, it can be quite difficult to use a control's properties.

## Finding property information

The standard way to get information about a control's properties is to refer to the documentation for the VBX control. Failing that, *TVbxControl* provides a number of methods for obtaining information about a control's properties.

You can find the total number of properties for a control by calling the *TVbxControl* member function *GetNumProps*, which returns an **int** that gives the total number of properties. These properties are indexed from 0 to the return value of *GetNumProps* –1.

You can find the name of any property in this range by calling the *TVbxControl* member function *GetPropName*. *GetPropName* takes one parameter, an **int** index number, and returns a string containing the name of the property.

Conversely, you can find the index of an property by calling the *TVbxControl* member function *GetPropIndex*. *GetPropIndex* takes one parameter, a string containing the property name, and returns the corresponding **int** property index.

## Getting control properties

You can get the value of a control property using either its name or its index number. Although using the index is somewhat more efficient (because there's no need to look up a string), using the property name is usually more intuitive. You can use either method, depending on your preference.

*TVbxControl* provides the function *GetProp* to get the properties of a control. *GetProp* is overloaded to allow getting properties using the index or name of the property. Each of these versions is further overloaded to allow getting a number of different types of properties:

```
// get properties by index
bool GetProp(int propIndex, int& value, int arrayIndex = -1);
bool GetProp(int propIndex, long& value, int arrayIndex = -1);
bool GetProp(int propIndex, HPIC& value, int arrayIndex = -1);
bool GetProp(int propIndex, float& value, int arrayIndex = -1);
bool GetProp(int propIndex, string& value, int arrayIndex = -1);

// get properties by name
bool GetProp(const char far* name, int& value, int arrayIndex = -1);
bool GetProp(const char far* name, long& value, int arrayIndex = -1);
bool GetProp(const char far* name, HPIC& value, int arrayIndex = -1);
bool GetProp(const char far* name, float& value, int arrayIndex = -1);
bool GetProp(const char far* name, string& value, int arrayIndex = -1);
```

In the versions where the first parameter is an **int**, you specify the property by passing in the property index. In the versions where the first parameter is a **char** *, you specify the property by passing in the property name.

Instead of returning the value property as the return value of the *GetProp* function, the second parameter of the function is a reference to the property's data type. Create an object of the same type as the property and pass a reference to the object in the *GetProp* function. When *GetProp* returns, the object contains the current value of the property.

The third parameter is the index of an array property, which you should supply if required by your control. You can find whether you need to supply this parameter and

the required values by consulting the documentation for your VBX control. The function ignores this parameter if it is –1.

### Setting control properties

As when you *get* control properties, you *set* the value of control property using either their name or their index number. Although using the index is somewhat more efficient (because there's no need to look up a string), using the property name is usually more intuitive. You can use either method, depending on your preference.

*TVbxControl* provides the function *SetProp* to set the properties of a control. *SetProp* is overloaded to allow setting properties using the index or name of the property. Each of these versions is further overloaded to allow setting a number of different types of properties:

```
// set properties by index
bool SetProp(int propIndex, int value, int arrayIndex = -1);
bool SetProp(int propIndex, long value, int arrayIndex = -1);
bool SetProp(int propIndex, HPIC value, int arrayIndex = -1);
bool SetProp(int propIndex, float value, int arrayIndex = -1);
bool SetProp(int propIndex, const string& value, int arrayIndex = -1);
bool SetProp(int propIndex, const char far* value, int arrayIndex = -1);

// set properties by name
bool SetProp(const char far* name, int value, int arrayIndex = -1);
bool SetProp(const char far* name, long value, int arrayIndex = -1);
bool SetProp(const char far* name, HPIC value, int arrayIndex = -1);
bool SetProp(const char far* name, float value, int arrayIndex = -1);
bool SetProp(const char far* name, const string& value, int arrayIndex = -1);
bool SetProp(const char far* name, const char far* value, int arrayIndex = -1);
```

In the versions where the first parameter is an **int**, you specify the property by passing in the property index. In the versions where the first parameter is a **char** *, you specify the property by passing in the property name.

The second parameter is the value to which the property should be set.

The third parameter is the index of an array property, which you should supply if required by your control. You can find whether you need to supply this parameter and the required values by consulting the documentation for your VBX control. The function ignores this parameter if it is –1.

Although there are *five* different data types you can pass in to *GetProp*, *SetProp* provides for *six* different data types. This is because the last two versions use both a **char** * and the ANSI *string* class to represent a string. This provides you with more flexibility when you're passing a character string into a control. In the *GetProp* version, casting is provided to allow a **char** * to function effectively as a *string* object.

## VBX control methods

Methods are functions contained in each VBX control that you can use to call into the control and cause an action to take place. *TVbxControl* provides compatibility with the methods contained in Visual Basic 1.0-compatible controls:

```
Move(int x, int y, int w, int h);
Refresh();
AddItem(int index, const char far *item);
RemoveItem(int index);
```

where:

- The *Move* function moves the control to the coordinates *x*, *y* and resizes the control to *w* pixels wide by *h* pixels high.
- The *Refresh* function refreshes the control's display area.
- The *AddItem* function adds the item *item* to the control's list of items and gives the new item the index number *index*.
- The *RemoveItem* function removes the item with the index number *index*.

# Chapter 17

# ObjectWindows dynamic-link libraries

A dynamic-link library (DLL) is a library of functions, data, and resources whose references are resolved at run time rather than at compile time.

Applications that use code from static-linked libraries attach copies of that code at link time. Applications that use code from DLLs share that code with all other applications using the DLL, therefore reducing application size. For example, you might want to define complex windowing behavior, shared by a group of your applications, in an ObjectWindows DLL.

This chapter describes how to write and use ObjectWindows DLLs.

## Writing DLL functions

When you write DLL functions that will be called from an application, keep these things in mind:

- Calls to 16-bit DLL functions should be made far calls. Similarly, pointers that are specified as parameters and return values should be made far pointers. You need to do this because a 16-bit DLL has different code and data segments than the calling application. (This isn't necessary for 32-bit DLLs.) Use the _FAR macro to make your code portable between platforms.
- Static data defined in a 16-bit DLL is global to all calling applications because 16-bit DLLs have one data segment that all 16-bit DLL instances share. Global data set by one caller can be accessed by another. If you need data to be private for a given caller of a 16-bit DLL, you need to dynamically allocate and manage the data yourself on a per-task basis. For 32-bit DLLs, static data is private for each process.

## DLL entry and exit functions

Windows requires that two functions be defined in every DLL: an entry function and an exit function. For 16-bit DLLs, the entry function is called *LibMain* and the exit function is called *WEP* (Windows Exit Procedure). *LibMain* is called by Windows for the first application that calls the DLL, and *WEP* is called by Windows for the last application that uses the DLL.

For 32-bit DLLs, *DllEntryPoint* serves as both the entry and exit functions. *DllEntryPoint* is called each time the DLL is loaded or unloaded, each time a process attaches to or detaches from the DLL, and each time a thread within a process is created or destroyed.

Windows calls the entry procedure (*LibMain* or *DllEntryPoint*) once, when the library is first loaded. The entry procedure initializes the DLL; this initialization depends almost entirely on the particular DLL's function, but might include the following tasks:

- Unlocking the data segment with UnlockData, if it has been declared as MOVEABLE
- Setting up global variables for the DLL, if it uses any

There is no need to initialize the heap because the DLL startup code (C0D*x*.OBJ) initializes the local heap automatically. The following sections describe the DLL entry and exit functions for 16- and 32-bit applications.

### LibMain

The 16-bit DLL entry procedure, *LibMain*, is defined as follows:

```
int FAR PASCAL LibMain(HINSTANCE hInstance,
                       uint16 wDataSeg,
                       uint16 cbHeapSize,
                       LPSTR lpCmdLine)
```

The parameters are described as follows:

- *hInstance* is the instance handle of the DLL.
- *wDataSeg* is the value of the data segment (DS) register.
- *cbHeapSize* is the size of the local heap specified in the module definition file for the DLL.
- *lpCmdLine* is a far pointer to the command line specified when the DLL was loaded. This is almost always null, because typically DLLs are loaded automatically without parameters. It is possible, however, to supply a command line to a DLL when it is loaded explicitly.

The return value for *LibMain* is either 1 (successful initialization) or 0 (unsuccessful initialization). Windows unloads the DLL from memory if 0 is returned.

### WEP

*WEP* is the exit procedure of a DLL. Windows calls it prior to unloading the DLL. This function isn't necessary in a DLL (because the Borland C++ run-time libraries provide a default one), but can be supplied by the DLL writer to perform any cleanup before the

DLL is unloaded from memory. Often the application has terminated by the time *WEP* is called, so valid options are limited.

Under Borland C++, *WEP* doesn't need to be exported. Here is the *WEP* prototype:

```
int FAR PASCAL WEP (int nParameter);
```

*nParameter* is either WEP_SYSTEMEXIT, which means Windows is shutting down, or WEP_FREE_DLL, which means just this DLL is unloading. *WEP* returns 1 to indicate success. Windows currently doesn't use this return value.

### DllEntryPoint

The 32-bit DLL entry point, *DllEntryPoint*, is defined as follows:

```
bool WINAPI DllEntryPoint(HINSTANCE hInstDll, uint32 fdwReason, LPVOID lpvReserved);
```

The parameters are described as follows:

- *hInstDll* is the DLL instance handle.
- *fdwReason* is a flag that describes why the DLL is being called (either a process or thread). The flags can take the following values:
  - DLL_PROCESS_ATTACH
  - DLL_THREAD_ATTACH
  - DLL_THREAD_DETACH
  - DLL_PROCESS_DETACH
- *lpvReserved* specifies further aspects of the DLL initialization and cleanup based on the value of *fdwReason*.

## Exporting DLL functions

After writing your DLL functions, you must export the functions that you want to be available to a calling application. There are two steps involved: compiling your DLL functions as exportable functions and exporting them. You can do this in the following ways:

- If you flag a function with the **_export** keyword, it's compiled as exportable and is then exported.
- If you add the **_export** keyword to a class declaration, the entire class (data and function members) is compiled as exportable and is exported.
- If you don't flag a function with **_export**, use the appropriate compiler switch or IDE setting to compile functions as exportable. Then list the function in the module definition (.DEF) file EXPORTS section.

## Importing (calling) DLL functions

You call a DLL function from an application just as you would call a function defined in the application itself. However, you must import the DLL functions that your application calls.

To import a DLL function, you can

- Add an IMPORTS section to the calling application's module definition (.DEF) file and list the DLL function as an import.
- Link an import library that contains import information for the DLL function to the calling application. (Use IMPLIB to make the import library).
- Explicitly load the DLL using *LoadLibrary* and obtain function addresses using *GetProcAddress*.

When your application executes, the files for the called DLLs must be in the current directory, on the path, or in the Windows or Windows system directory; otherwise your application won't be able to find the DLL files and won't load.

# Writing shared ObjectWindows classes

A class instance in a DLL can be shared among multiple applications. For example, you can share code that defines a dialog box by defining a shared dialog class in a DLL. To share a class, you need to export the class from the DLL and import the class into your application.

## Defining shared classes

To define shared classes, you need to

- Conditionally declare your class as either **_export** or **_import**.
- Pass a *TModule** parameter to the window constructors (in some situations).

**Note** If you declare a shared class in an include file that is included by both the DLL and an application using the DLL, the class must be declared **_export** when compiling the DLL and **_import** when compiling the application. You can do this by defining a group of macros, one of which is conditionally set to **_export** when building the DLL and to **_import** when using the DLL. For example,

```
#if defined(BUILDEXAMPLEDLL)
  #define _EXAMPLECLASS __export
#elif defined (USEEXAMPLEDLL)
  #define _EXAMPLECLASS __import
#else
  #define _EXAMPLECLASS
#endif

class _EXAMPLECLASS TColorControl : public TControl
{
  public:
    ⋮
};
```

By defining BUILDEXAMPLEDLL (on the command line, for example) when you are building the DLL, you cause _EXAMPLECLASS to expand to **_export**. This causes the class to be exported and shared by applications using the DLL.

By defining USEEXAMPLEDLL when you're building the application that will use the DLL, you cause _EXAMPLECLASS to expand to **_import**. The application will know what type of object it will import.

# The TModule object

An instance of the *TModule* class serves as the object-oriented interface for an ObjectWindows DLL. *TModule* member functions provide support for window and memory management, and process errors. See the *ObjectWindows Reference Guide* for a complete *TModule* class description.

The following code example shows the declaration and initialization of a *TModule* object. This example is conditionalized so that either 16-bit (*LibMain*) or 32-bit (*DllEntryPoint*) DLLs can use the same source file.

```
static TModule *ResMod;

#if defined(__WIN32__)
  bool WINAPI
  DllEntryPoint(HINSTANCE instance, uint32 /*flag*/, LPVOID)
#else   // !defined(__WIN32__)
  int
  FAR PASCAL
  LibMain(HINSTANCE instance,
          uint16 /*wDataSeg*/,
          uint16 /*cbHeapSize*/,
          char far* /*cmdLine*/)
#endif
  {
    // We're using the DLL and want to use the DLL's resources
    //
    if (!ResMod)
      ResMod = new TModule(0,instance);
    return true;
  }
```

Within the entry point function, the *TModule* object *ResMod* is initialized with the instance handle of the DLL. If the module isn't loaded an exception is thrown.

If your DLL requires additional initialization and cleanup, you can perform this processing in your *LibMain*, *DllEntryPoint*, or *WEP* functions. A better method, though, is to derive a *TModule* class, define data members for data global to your DLL within the class, and perform the required initialization and cleanup in its constructor and destructor.

After you've compiled and linked your DLL, use IMPLIB to generate an import library for your DLL. This import library will list all exported member functions from your shared classes as well as any ordinary functions you've exported.

# Using ObjectWindows as a DLL

To enable your ObjectWindows applications to share a single copy of the ObjectWindows library, you can dynamically link them to the ObjectWindows DLL. To do this, you'll need to be sure of the following:

- When compiling, define the macro _OWLDLL on the compiler command line or in the IDE.
- Instead of specifying the static link ObjectWindows library when linking (that is, OWLWS.LIB, OWLWM.LIB, OWLWL.LIB, or OWLWF.LIB), specify the ObjectWindows DLL import library (OWLWI.LIB for 16-bit applications, or OWLWFI.LIB for 32-bit applications).

# Calling an ObjectWindows DLL from a non-ObjectWindows application

When a child window is created in an ObjectWindows DLL, and the parent window is created in an ObjectWindows application, the ObjectWindows support framework for communication between the parent and child windows is in place. But you can also prepare your DLL for use by non-ObjectWindows applications.

When a child window is created in an ObjectWindows DLL and the parent window is created by a non-ObjectWindows application, the parent-child relationship must be simulated in the ObjectWindows DLL. This is done by constructing an alias window object in the ObjectWindows DLL that is associated with the parent window whose handle is specified on a DLL call.

In the following code, the exported function *CreateDLLWindow* is in an ObjectWindows DLL. The function will work for both ObjectWindows and non-ObjectWindows applications.

```
bool far _export
CreateDLLWindow(HWND parentHWnd)
{
  TWindow* parentAlias = GetWindowPtr(parentHWnd);  // check if an OWL window

  if (!parentAlias)
    parentAlias = new TWindow(parentHWnd); // if not, make an alias

  TWindow* window = new TWindow(parentAlias, "Hello from a DLL!");
  window->Attr.Style |= WS_POPUPWINDOW | WS_CAPTION | WS_THICKFRAME
                      | WS_MINIMIZEBOX | WS_MAXIMIZEBOX;
  window->Attr.X = 100; window->Attr.Y = 100;
  window->Attr.W = 300; window->Attr.H = 300;
  return window->Create();
}
```

*CreateDLLWindow* determines if it has been passed a non-ObjectWindows window handle by the call to *GetWindowPtr*, which returns 0 when passed a non-ObjectWindows

window handle. If it is a non-ObjectWindows window handle, an alias parent *TWindow* object is constructed to serve as the parent window.

# Implicit and explicit loading

Implicit loading is done when you use a .DEF or import library to link your application. The DLL is loaded by Windows when the application using the DLL is loaded.

Explicit loading is used to load DLLs at run time, and requires the use of the Windows API functions *LoadLibrary* to load the DLL and *GetProcAddress* to return DLL function addresses.

# Mixing static and dynamic-linked libraries

The ObjectWindows libraries are built using the BIDS (container class) libraries, which in turn are built using the C run-time library.

If you link with the DLL version of the ObjectWindows libraries, you must link with the DLL version of the BIDS and run-time libraries. You do this by defining the _OWLDLL macro. This isn't the only combination of static and dynamic-linked libraries you can use: each line in the table below lists an allowable combination of static and dynamic-linked libraries.

**Table 17.1** Allowable library combinations

| Static libraries | Dynamically linked libraries |
|---|---|
| OWL, BIDS, RTL | (none) |
| OWL, BIDS | RTL |
| OWL | BIDS, RTL |
| (none) | OWL, BIDS, RTL |

Chapter 18

# Support for OLE in Borland C++

This section of the *ObjectWindows Programmer's Guide* describes ObjectComponents, a set of classes for creating OLE 2 applications in C++. This introductory chapter answers these questions:

- What does ObjectComponents do?
- What is OLE?
- What does OLE look like?
- What is ObjectComponents?
- How does ObjectComponents work?
- What documentation and tools help with using ObjectComponents?
- What do all the terms mean?

Subsequent chapters show how to create different kinds of programs using ObjectComponents.

## What does ObjectComponents do?

ObjectComponents makes OLE programming easy. It supports all the following OLE 2 capabilities:

- Linking
- In-place editing
- OLE Clipboard operations
- Automation servers and controllers
- Localization
- Embedding
- Drag-and-drop operations
- Compound document storage
- DLL servers
- Registration

These features are described in more detail in "OLE 2 features supported by ObjectComponents" on page 276.

Using ObjectComponents confers these other benefits as well:

- An easy upgrade path to linking and embedding for existing C++ applications, especially if they use ObjectWindows
- Easy automation of existing C++ applications, whether or not they use ObjectWindows
- Default implementations of standard OLE 2 user interface elements, such as the Insert Object and Paste Link dialog boxes
- The ability to create OLE 2 applications with AppExpert, which generates and understands the new OLE classes
- Compatibility with other OLE applications, including OLE 1 applications, whether or not they were built with Borland tools
- Virtually no operating overhead imposed on ObjectWindows applications that choose not to use OLE

# Where should you start?

That depends on what you want to know and what application you want to create. This section lists things you might want to do and tells you where to find the information you need.

## Writing applications

The right starting place depends on whether you are creating a new application or adapting an existing one.

### Creating a new application

You can generate a complete OLE application almost instantly using AppExpert. AppExpert fully supports all the new features of ObjectComponents. To start an OLE application from scratch, simply choose Project | AppExpert from the IDE menu. For more information about AppExpert, consult the *User's Guide*.

The programs AppExpert creates use the ObjectWindows Library. If you are new to ObjectWindows, begin with the *ObjectWindows Tutorial* book.

### Converting an application into an OLE container

Where you should start depends on whether your application uses ObjectWindows, and if so, whether it uses the Doc/View model.

**Table 18.1** How to add container support to an existing application

| ObjectWindows? | Doc/View? | Where to start |
|---|---|---|
| Yes | Yes | "Turning a Doc/View application into an OLE container" on page 303 |

**Table 18.1** How to add container support to an existing application (continued)

| ObjectWindows? | Doc/View? | Where to start |
|---|---|---|
| Yes | No | "Turning an ObjectWindows application into an OLE container" on page 315 |
| No | No | "Turning a C++ application into an OLE container" on page 328 |

## Converting an application into a linking and embedding server

Where you should start depends on whether your application uses ObjectWindows, and if so, whether it uses the Doc/View model.

**Table 18.2** How to add server support to an existing application

| ObjectWindows? | Doc/View? | Where to start |
|---|---|---|
| Yes | Yes | "Turning a Doc/View application into an OLE server" on page 341 |
| Yes | No | "Turning an ObjectWindows application into an OLE server" on page 352 |
| No | No | "Turning a C++ application into an OLE server" on page 359 |

## Adding automation support

The process of adding automation support to an existing application is the same regardless of whether the application uses ObjectWindows or the Doc/View model. For help creating an automation server, a controller, or a type library, turn to the indicated section.

- **Automation server**: "Automating an application" on page 381
- **Automation client**: "Creating an automation controller" on page 407
- **Type library**: "Creating a type library" on page 405

## Other useful topics

Here are some topics common to different kinds of OLE applications. For help with them, turn to the indicated section.

- **DLL server**: "Making a DLL server" on page 374
- **Localization**: "Localizing symbol names" on page 397
- **Registration**: "Building registration tables" on page 344
- **Compiling and linking**: "Building an ObjectComponents application" on page 286
- **Exception handling**: "Exception handling in ObjectComponents" on page 284

# Learning about ObjectComponents

The tasks listed in this section help you find your way around ObjectComponents.

- **Understanding OLE**

  For an introduction to OLE, see the following section, "What is OLE?" For illustrations showing how common OLE interactions look onscreen, see "What does OLE look like?" on page 267.

- **Surveying the new classes**
  For tables summarizing new classes and messages in ObjectComponents and ObjectWindows, see "Using ObjectComponents" on page 278.
- **Understanding how** ObjectComponents **works**
  "How ObjectComponents works" on page 287 explains how ObjectComponents classes mediate between OLE and your own C++ classes.
- **Finding example programs**
  Brief description of some of the sample ObjectComponents programs in Borland C++ appear in "Example programs" on page 294.
- **Finding the right documentation**
  All the parts of the documentation that describe ObjectComponents are listed in "Books" on page 293 and "Online Help" on page 294.
- **Understanding terms**
  For definitions of terms used in the documentation, see the "Glossary of OLE terms" on page 295. Skimming the glossary is also a good way to introduce yourself to the features of ObjectComponents.

# What is OLE?

OLE, which stands for object linking and embedding, is an operating system extension that lets applications achieve a high degree of integration. OLE defines a set of standard interfaces so that any OLE program can interact with any other OLE program. No program needs to have any built-in knowledge of its possible partners.

Programmers implement OLE applications by creating objects that conform to the Component Object Model (COM). COM is the specification that defines what an OLE object is. COM objects support interfaces, composed of functions for other objects to call. OLE defines a number of standard interfaces. COM objects intended for public access expose their interfaces in a registration database. Interfaces have unique identifiers to distiguish them.

ObjectComponents encapsulates the COM specification for creating objects and provides default implementations of the interfaces used for two common OLE tasks: linking and embedding, and automation. *Linking and embedding* lets one application incorporate live data from other OLE applications in its documents. *Automation* lets one application issue commands to control another application.

## Linking and embedding

*Linking* and *embedding* refer to the transfer of data from one program to another. The first program, the *server*, sends its data to the second program, the *container*. For example, cells from a spreadsheet can be dropped into a word processing document. Of course you don't need OLE to pass data from one Windows program to another. You can do that much with just the Clipboard. The difference between OLE and the Clipboard is that in OLE the receiving program doesn't have to know anything at all about the format of the data in the object. Any OLE server application can give its data to any OLE

container application. Thanks to OLE, the container doesn't care whether the object it receives is a metafile, a bitmap, or ASCII text. The server passes whatever data it uses internally and the container accepts it. Furthermore, the object remains dynamic even after being transplanted. When the container wants to display, modify, or save the object, it calls OLE to do it. OLE, working behind the scenes, calls the server to execute the user's command. The object belongs to the container's document, but OLE maintains a live connection back to the server. The user can continue to edit the object using all of the server's tools. As a result, the user can combine objects from different servers into a single document without losing the ability to update and modify any object as the document evolves.

## Automation

*Automation* happens when one program issues commands to another. If you write a calculator program, for example, you might allow other programs to issue commands like these:

Press the nine button.
Press the plus button.
Press the six button.
Press the equals button.
Tell me what's in the Total window.

These are commands a person might normally issue through the calculator's user interface. With automation, the calculator exposes its internal functions to other programs. The calculator becomes an *automation object*, and programs that send commands to it are *automation controllers*. OLE defines standard interfaces that let a controller ask any installed server to create one of its objects. OLE also makes it possible for the controller to browse through a list of automated commands the server supports and execute them.

# What does OLE look like?

The linking and embedding features of OLE include a standard user interface for performing common operations such as placing OLE objects in container documents and activating them once they are linked or embedded. The OLE standards cover menu commands, dialog boxes, tool bars, drag and drop support, and painting conventions, so that the user interface for OLE operations is consistent across applications. Together, ObjectComponents and ObjectWindows execute most of the interface tasks for you.

Understanding OLE programming can be difficult without a clear grasp of the interface you are trying to create. The following sections present pictures of a container showing what happens onscreen at each step in a common sequence of OLE operations. The user runs a container, inserts objects from several OLE servers into the document, edits an

object, and saves the document. The descriptions of these steps introduce the following OLE features:

- Insert Object command
- Embedding
- Object verbs
- Activating an object
- In-place editing
- Tool bar negotiation
- Paste Special command
- Linking
- Selecting an object
- Open editing
- Menu merging

## Inserting an object

The first illustration shows the example program called SdiOle, which is an OLE container using the single-document interface (SDI) and written with ObjectWindows and ObjectComponents. The source code for SdiOle is in EXAMPLES/OWL/OCF/SDIOLE.

The SdiOle Edit menu contains five standard OLE commands that most containers possess: Paste Special, Paste Link, Insert Object, Links, and Object. Most of these are disabled in the illustration because the Clipboard is empty and nothing has been linked or embedded in the open SdiOle document.

ObjectWindows implements all five of the standard commands for you if you like, but a container does not have to use them all. This section explains only the Insert Object and Object commands. For a brief summary of all the commands, see Table 19.3 on page 312.

**Figure 18.1** The Edit menu in the sample program SdiOle

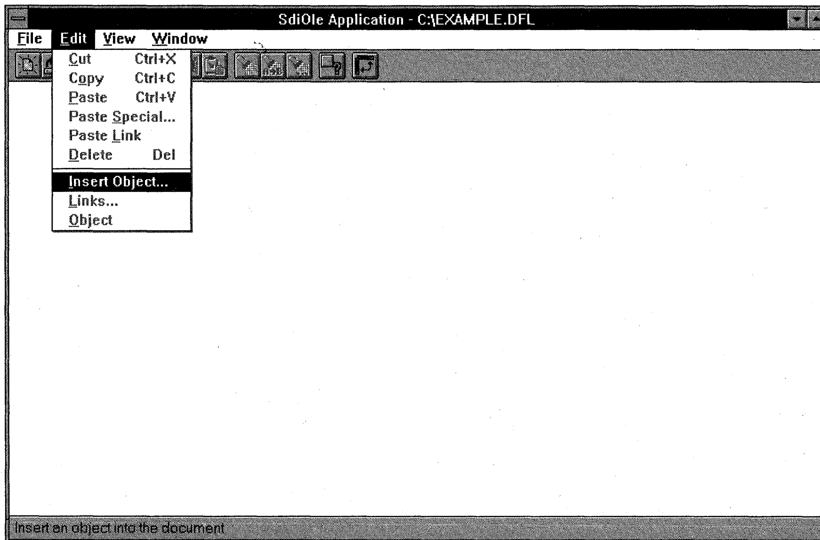


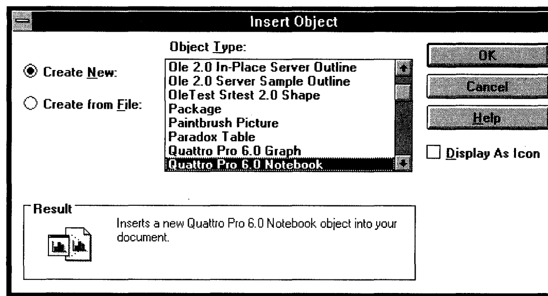

Figure 18.2 shows the Insert Object dialog box. Like the common dialog boxes in Windows for opening files and choosing fonts, the Insert Object dialog box is a standard resource implemented by the system. For consistency, it is best to use the standard

dialog boxes unless your application has some unusual requirement that the standard dialog box does not meet.

The box under Object Type lists all the kinds of objects available in the system. Whenever a server installs itself, it tells the system what objects it can create. The system keeps this information in its registration database. The Insert Object dialog box queries the database and shows all the types that OLE can create for you using the available server applications.

In the illustration, the user has chosen to insert a Quattro Pro spreadsheet. The Result box at the bottom of the dialog box explains what will happen if the user clicks OK now. Because the Create New button is selected, clicking OK will embed a new, empty spreadsheet object into the user's open document. Figure 18.3 shows the result. (The Create from File button is explained later.)

**Figure 18.2** The Insert Object dialog box




## Editing an object in place

In Figure 18.3 the SdiOle application window is barely recognizable. Only from the title bar at the top of the window can you be sure this is the same application. The menu bar has changed—it has many more drop-down menus than it did in Figure 18.1. The Block, Notebook, Graphics, Tools, and Help menus are all new. There are three button bars now instead of one, and none of them is the same as the original one.

All the new window elements come from Quattro Pro, the server application that created the active object. Quattro Pro has taken over the SdiOle window and is displaying its own menus and tool bars. All the Quattro Pro menu and tool bar commands can be executed right here in SdiOle. The feature of OLE that lets a server take over a container's main window is called *in-place editing*. It lets the user edit the object in its place, without switching back and forth between different windows. The programming task that makes this possible is called *menu merging*, combining menus from two programs in one menu bar.

**Figure 18.3** A newly inserted object being edited in place

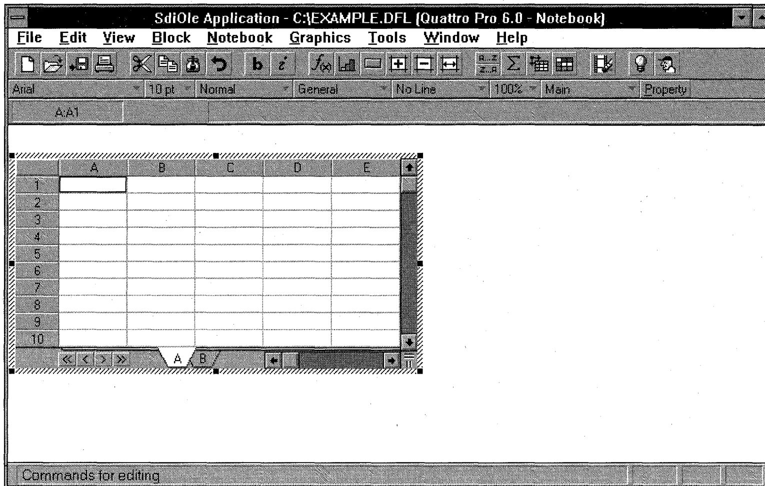
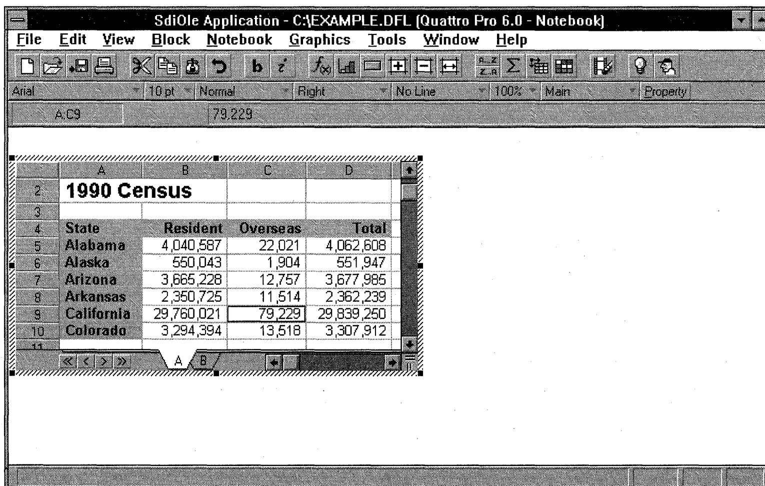



Figure 18.4 shows what the object looks like after it is edited. The user entered labels and numbers into notebook cells and created a small spreadsheet. Although many programs let you paste data from other programs into your documents, without OLE you cannot continue to edit the objects after they are transferred.

**Figure 18.4** The same inserted object after being edited




The user who entered the data shown in Figure 18.4 clearly had access to Quattro Pro tools. SdiOle is a very simple application and knows nothing about columns and rows or fonts and shading. But even though the Quattro Pro server created and formatted the object, that data in the object belongs to the container. When the user chooses File | Save from the SdiOle menu bar, what gets written is an SdiOle document, not a Quattro Pro document. With the help of ObjectComponents and the OLE system, SdiOle marks an area in its own file to store the data for the embedded object. When the user chooses

File | Load to reload the same document, the spreadsheet cells will still be there. If the user tries to edit the object again, OLE invokes Quattro Pro to take over the SdiOle window once more. The object remains associated with the application that created it even though the object is stored in a foreign file.

When OLE places the data for an object directly into the container's document as it has the data for this spreadsheet, the object is said to be *embedded*. Besides embedding, OLE also *links* objects to container documents, as you'll see in Figure 18.7.

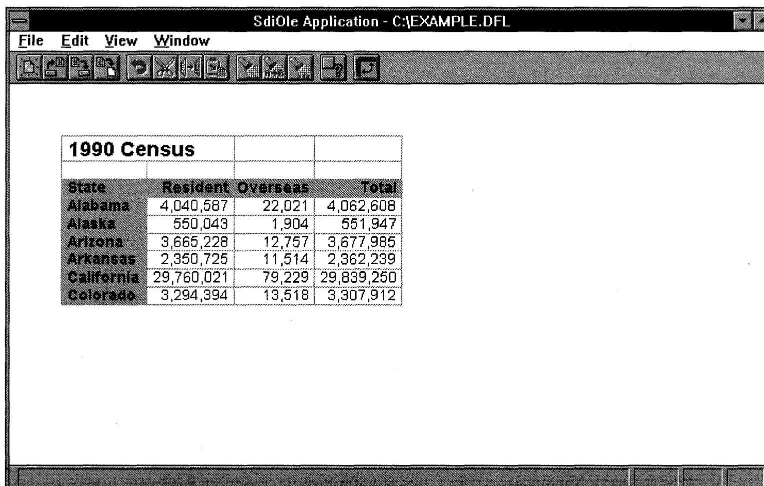
## Activating, deactivating, and selecting an object

In Figures 18.3 and 18.4, the embedded object is outlined by a thick gray rectangle. The presence of this rectangle indicates that the object is active. The activation rectangle appears when you double-click the object. Usually activating an object initiates an editing session, but the server decides whether to follow that convention. For example, embedded sound objects might play when activated. In most cases, only one object can be active at a time.

The activation rectangle in Figure 18.4 has eight small black boxes spaced evenly around it. They are called *grapples*. The user can resize the object by clicking a grapple and dragging the mouse. Also, the user can move the object by clicking anywhere else on the activation rectangle and dragging. ObjectWindows uses the *TUIHandle* class to draw rectangles and grapples around objects.

Figure 18.5 shows what happens in the container window when the user clicks the mouse button outside the activated object. The activation rectangle goes away. The object is now *inactive*. Deactivating an object tells OLE that you are through editing. The server relinquishes its place, and the container's window returns to normal. The only commands on the menu bar are the ones SdiOle put there. The tool bar and window caption are back to normal, as well.

**Figure 18.5** The container's restored user interface after the object becomes inactive




| State | Resident | Overseas | Total |
|---|---|---|---|
| Alabama | 4,040,587 | 22,021 | 4,062,608 |
| Alaska | 550,043 | 1,904 | 551,947 |
| Arizona | 3,665,228 | 12,757 | 3,677,985 |
| Arkansas | 2,350,725 | 11,514 | 2,362,239 |
| California | 29,760,021 | 79,229 | 29,839,250 |
| Colorado | 3,294,394 | 13,518 | 3,307,912 |

You can *select* an inactive object without activating it. When you press the mouse button over an inactive object, the container draws a thin black rectangle to show that you have selected it. The selection rectangle is visible in Figure 18.6. Like the activation rectangle, it has grapples. The user can move and resize a selected object just like an active object.

## Finding an object's verbs

When an object is selected, like the one in Figure 18.6, the container modifies its menus to offer a choice of whatever actions the object's server can do with the object. OLE calls these actions *verbs*. Conventionally, the container displays available verbs in two places: on its Edit menu and on a SpeedMenu. The SpeedMenu in Figure 18.6 popped up when the user right-clicked the object. The first three commands on the SpeedMenu are always Cut, Copy, and Delete. The fourth item, Notebook Object, changes depending on the object selected. When an object from Paradox is inserted, for example, the fourth item becomes Paradox 5 Object.

The smaller cascading menu lists the particular verbs that the server supports. Quattro Pro has only two verbs. It can edit an object or open an object. The Edit verb initiates an in-place editing session, as shown in Figure 18.3. The Open verb inititates an open editing session, as shown in Figure 18.9.

The final item, Convert, is the same for all objects. It invokes another standard OLE dialog box that lets the user convert an object from one server's data format to another. The Convert command is useful when, for example, you have Paradox installed on your machine, but someone gives you a compound document with an embedded object from some other database application. If Paradox knows how to convert data from the other database, then the Convert command binds the foreign database object to Paradox.

**Figure 18.6** The speed menu for a selected object

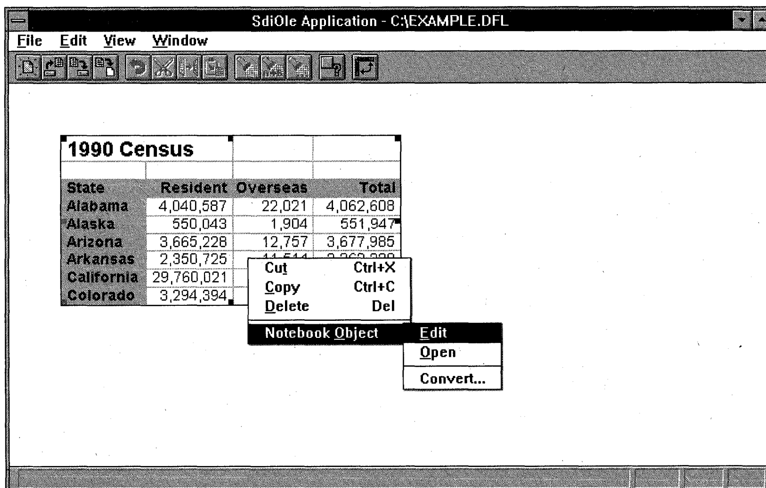



Figure 18.8 shows where verbs appear on the Edit menu. When no object is selected, the last command on the Edit menu is disabled and says simply Object, as you see in Figure

18.1. When an object is selected, the Object command changes to describe the selected object. In the example, Object changes to Notebook Object.

## Linking an object

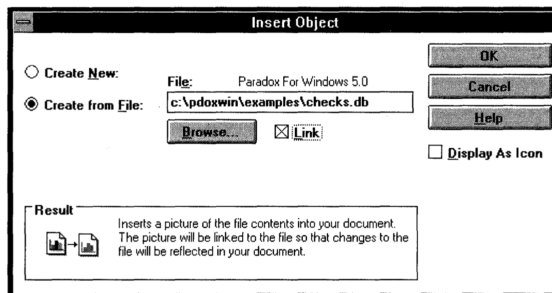
By default, the Insert Object command creates a brand new empty object, like the one in Figure 18.3, and embeds it. Instead of embedding an object, you can choose to link it.

When OLE links an object, it does not store the object's data in the container's document. It stores only the name of the server file where the data is stored (along with the location of the data within the file and a snapshot of the object as it appears onscreen. The snapshot is usually a metafile.) The container doesn't receive a copy of the object; it receives a pointer to the object. OLE still draws the object in the container's document, just as though it was embedded, but the container doesn't own the data.

If the server document that holds the data for the linked object is deleted, then the user can no longer activate and edit the linked object. On the other hand, if the data in the server document is updated, then the updates appear automatically in all the container documents that have been linked to the same object. If several documents *embed* the same object, then they are creating copies, and changes made in one document have no effect on the copies in other documents.

Figure 18.7 shows what happens if you select the Create From File button in the Insert Object dialog box. Instead of creating a new empty object, you choose a file with existing data and OLE invokes the server that created the file. You can embed data from the file, but in Figure 18.7 the user has checked the Link box, so when the user clicks OK, OLE does not copy data from CHECKS.DB into the server's document. It creates a link that refers back to the data stored in the original file.

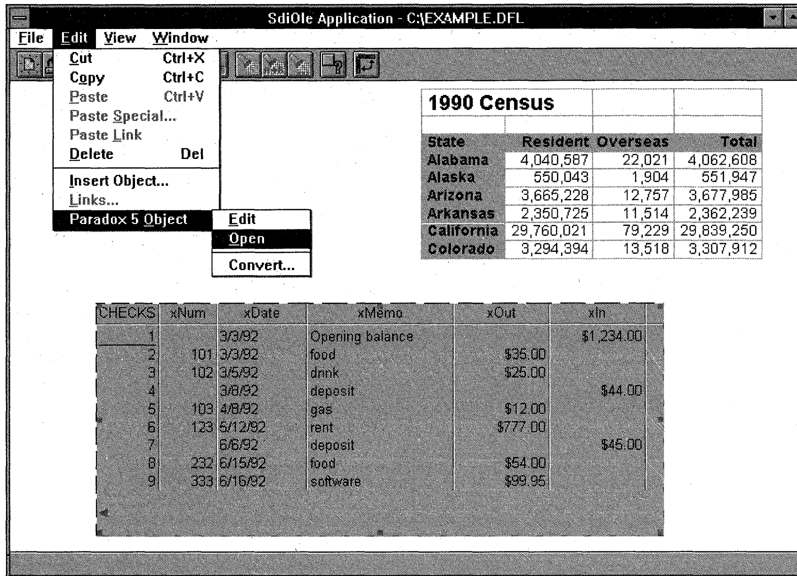
**Figure 18.7** The Insert Object dialog box just before inserting a linked object




The text in the Result box at the bottom of the dialog box explains what will happen when the user clicks OK. You can see the result in Figure 18.8. The EXAMPLE.DFL document now contains two OLE objects—the embedded Quattro Pro spreadsheet and the linked Paradox table.

Neither of the two objects is active. The spreadsheet is inactive and the database table is selected. Because the database table is linked, ObjectWindows draws the selection rectangle with a dashed line. Compare the selection rectangle in Figure 18.8 to the one for an embedded object in Figure 18.6.

**Figure 18.8** The new verb list for the newly linked object

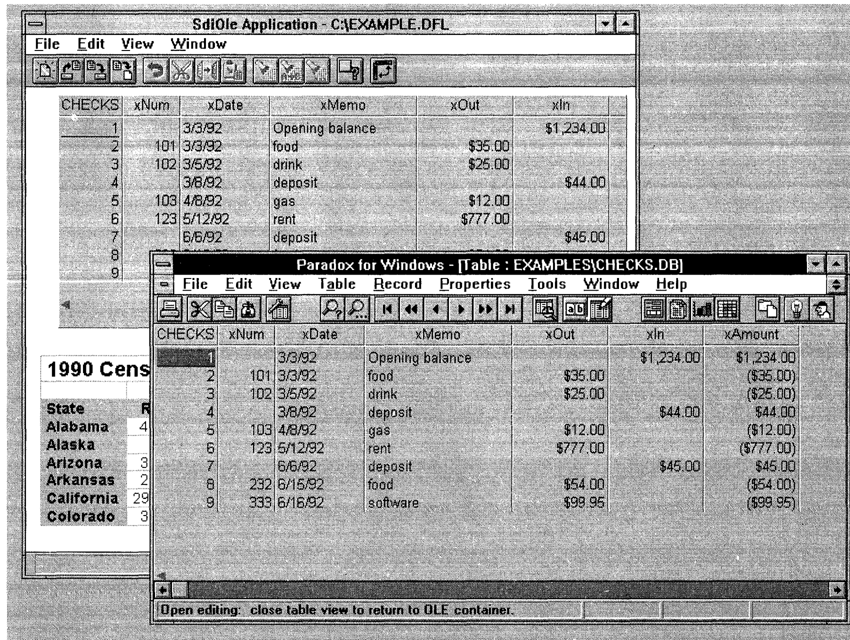



# Opening an object to edit it

The Edit menu in Figure 18.8 shows the verbs for the selected Paradox table. Edit and Open are the two most common verbs, and Quattro Pro and Paradox both use them. Choosing the Open verb produces the screen shown in Figure 18.9. The same table is visible in two windows—the container window where it is linked and the server window where it is being edited. When finished editing in the server window, the user chooses File | Close and returns to the container. Any changes made during the editing session automatically appear in the container window afterward.

Contrast this editing session with the in-place editing in Figure 18.3. In this session, the container window remains unchanged. The SdiOle window has only its own commands and its own tool bar. The editing takes place in a separate window that OLE opened just for this session. Returning to the server to edit is called *open editing*. Some servers support only open editing, not in-place editing.

**Figure 18.9** An object opened for editing




This series of illustrations shows the most common linking and embedding operations. The user links or embeds an object, selects it, activates it, edits it in place or open, and saves the compound document complete with its OLE object. The examples show how to link and embed objects with the Insert Object dialog box, but there are other ways as well. The Paste, Paste Special, and Paste Link commands can all create OLE objects from data on the Clipboard. You can also link or embed objects by dragging them from one applicaton and dropping them on another.

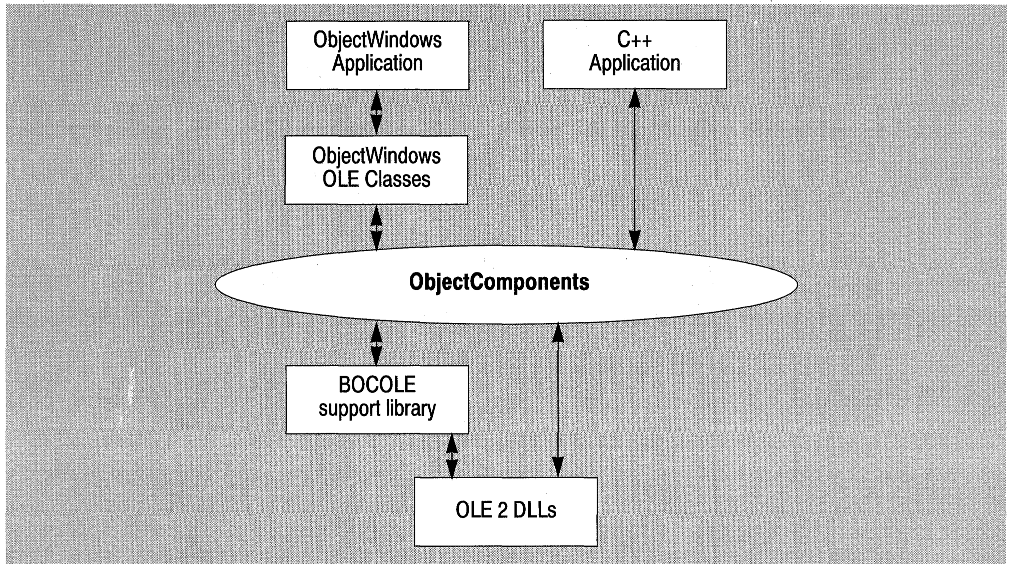
# What is ObjectComponents?

Microsoft's OLE 2 operating system extensions require the programmer to implement a variety of interfaces depending on the tasks an application undertakes. Borland has developed an OLE engine, already used in several of its commercial applications, that simplifies the programmer's job by implementing a smaller set of high-level interfaces on top of OLE. The engine resides in a library called BOCOLE.DLL. The BOCOLE support library provides default implementations for many standard OLE interfaces.

C++ programmers can make use of the OLE support in BOCOLE.DLL through a set of new classes collectively called the ObjectComponents Framework (OCF). Instead of implementing OLE-style interfaces, you create objects from the ObjectComponents classes and call their methods. Your own classes can gain OLE capabilities simply by inheriting from the ObjectComponents classes. ObjectComponents translates between C++ and OLE.

Figure 18.10 shows how the layers of Borland's OLE support fit together.

**Figure 18.10** How applications interact with OLE through ObjectComponents




The ObjectComponents classes implement OLE-style interfaces for talking to the BOCOLE support library. Your programs reach OLE by calling methods from ObjectComponents classes. When OLE sends information to you, ObjectComponents sends messages to your application using the standard Windows message mechanisms. The ObjectComponents classes also contain default implementations for all the OLE messages. You can override the default event handlers selectively to modify your application's responses.

ObjectComponents is not part of the ObjectWindows Library. That means C++ programs that don't use ObjectWindows can still take full advantage of ObjectComponents for linking, embedding, and automation. But ObjectWindows can simplify your work even more. ObjectWindows 2.5 introduces new classes such as *TOleWindow* and *TOleDocument* that inherit from ObjectComponents classes to bring OLE support into Borland's C++ application framework. An ObjectWindows application that uses the Doc/View model doesn't need to use ObjectComponents directly at all. A few simple changes to your Doc/View program will have you linking and embedding almost instantly. Programs that don't use the Doc/View model can do the same thing with just a little more work.

The chapters that follow explain step by step how to modify your code to create containers, servers, automation objects, and controllers.

## OLE 2 features supported by ObjectComponents

The following list summarizes the OLE 2 capabilities that ObjectComponents gives your applications. The descriptions assume you are using ObjectWindows, as well. All the same features are available through ObjectComponents without ObjectWindows, but then you have to code explicitly some things that ObjectWindows does by default.

- **Linking and embedding**: To embed data from one application in the document of another, ObjectComponents gives you classes to represent the data in the object and an image of the data for drawing on the screen. The data must be separable from its graphical representation because in OLE transactions they are sometimes handled by different programs. When the container asks the server for an object to embed, the server must provide data and a view of the data. The server can also be asked to edit the object even after it is embedded and to read or write the object to and from the container's document file. The ObjectComponents classes handle both sides of these negotiations for you.
- **Clipboard operations**: The default event handlers for the ObjectComponents messages handle cutting and pasting for you. If you add to your menus standard commands such as Insert Object and Paste Link, ObjectComponents will implement them for you.
- **Drag and drop operations**: The default event handlers for ObjectComponents messages help you here, too. If the user drops an OLE object on your container's window, ObjectComponents inserts it in your document. If the user double-clicks the embedded object, ObjectComponents activates it. If the user drags the object, ObjectComponents moves it.
- **Standard OLE 2 user interface**: OLE defines standard user interface features and asks OLE programmers to comply with them. Built into ObjectComponents are dialog boxes for commands like Insert Object, Paste Special, and Paste Link; a pop-up menu that appears whenever the user right-clicks an embedded object; and an item on the container's Edit menu that always shows the verbs (server commands) available for the active object. ObjectComponents even arranges to modify the container's window if the server takes over the container's tool bar, status bar, and menus for in-place editing.
- **Compound files**: A new ObjectComponents class (*TOcStorage*) encapsulates file input and output to compound files. If you convert an ObjectWindows Doc/View application into an ObjectComponents container, the document writes itself to compound files automatically, creating storages and substorages within the file as needed. (Instructions for the conversion appear in Chapter 19.)
- **EXE and DLL servers**: ObjectComponents lets you construct your OLE server as either a standalone executable program or as an in-process DLL server. DLL servers respond to clients more quickly because a DLL is not a separate process. OLE doesn't have to serialize calls or marshall parameters to communicate between a DLL server and its client. See "Making a DLL server" on page 374 for more information.
- **Automation**: ObjectComponents permits C++ classes to be automated without structural changes to the classes themselves. It accomplishes this with nested classes that have direct access to the existing class members. These nested classes instantiate small command objects that reach the members through standard C++ mechanisms, avoiding the use of restrictive, non-portable stack manipulations. The command objects support hooks for undoing, recording, and filtering automation commands. A program can even send itself automation commands using standard C++ code. Chapters 21 and 22 describe automation programming.

- **Type libraries**: A type library describes for OLE all the classes, methods, properties, and data members available for controlling an automated application. Once you create an automation server (following the steps in Chapter 21), you can ask ObjectComponents to build and register the type library for you. Instructions for creating a type library are on page 405.
- **Registration**: OLE requires applications to register themselves with the system by providing a unique identifier string. For servers, this string and much other information besides must be recorded in the system's registration database as part of the program's installation process. With ObjectComponents, all you have to do is list all the information in one place using macros. Every time your server starts up, ObjectComponents confirms that the database accurately reflects the server's status. When necessary, ObjectComponents records or updates registration entries automatically. For more about registration, see "2. Registering a container" on page 306.
- **Localization**: OLE servers need to speak the language of their client programs. If an automation server is marketed in several countries, it needs to recognize commands sent in each different language. A linking and embedding server registers strings that describe its objects to the user, and those too should be available in multiple languages in order to accommodate whatever language the user might request. If you provide translations for your strings, ObjectComponents uses the right strings at the right time. Add your translations to the program's resources and mark the original strings as localized when you register them. At run time, ObjectComponents quickly and efficiently retrieves translations to match whatever language OLE requests. For more about localization, see "Registering localized entries" on page 373.

# Using ObjectComponents

This section includes information to help you use ObjectComponents. It surveys the classes and messages in ObjectComponents, as well as new classes in ObjectWindows that help you take advantage of ObjectComponents. It also explains how ObjectComponents uses C++ exception handling, how to build an ObjectComponents application, and what files to distribute with your application.

## Overview of classes and messages

The following tables introduce the ObjectComponents classes and messages you are likely to use most often. Subsequent chapters describe their use in more detail.

## Linking and embedding classes

The classes in Table 18.3 support linking and embedding, but if your program uses ObjectWindows you won't need to work directly with most of them.

**Table 18.3** Some ObjectComponents classes used for linking and embedding

| Class | Description |
|---|---|
| TOcApp | Connects containers and servers to OLE. It implements COM interfaces for the application. |
| TOcDocument | Represents a compound document. It holds *parts* (embedded objects). |
| TOcModule | A mix-in class for deriving the application object in a linking and embedding program. It coordinates some basic housekeeping chores related to registration and memory management. |
| TOcPart | Represents an embedded or linked object in a document. |
| TOcRegistrar | Records application information in the system registration database and tells OLE when the application starts and stops. Also creates the *TOcApp* object and responds when OLE wants a server to make something. |
| TOcRemView | Represents a remote view for a server document. The server creates a remote view for every object it donates to a container. The remote view is drawn in the container's window. |
| TOcView | Responsible for displaying a part. A container needs a view for every part it embeds. |

Although ObjectComponents includes classes for documents and views, it does not require applications to use the ObjectWindows Doc/View model. If you do use the Doc/View model, the new *TOleDocument* and *TOleView* classes make OLE programming even easier. ObjectWindows is not required, however. Any C++ program can use the ObjectComponents Framework. The chapters that follow address all types of applications.

## Connector objects

A few of the ObjectComponents classes actually implement COM interfaces. (COM stands for Component Object Model. COM is the standard that defines what an OLE object is.) Most of the supported interfaces are not standard OLE interfaces; they are custom interfaces that communicate with OLE through the BOCOLE support library. But like any COM object they do implement *IUnknown* (by deriving from *TUnknown*, as shown in Figure 18.11).

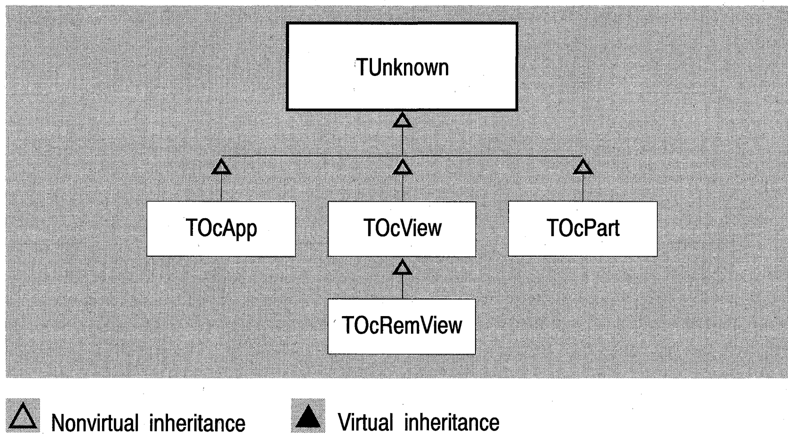
The classes that define COM objects for linking and embedding are *TOcApp*, *TOcView*, *TOcRemView*, and *TOcPart*. These classes are special because they connect your application to OLE. They are called *connector objects*. An ObjectComponents application must create connector objects in order to interact with other OLE applications.

Because they are COM objects, connector objects have one peculiarity: their destructors are protected so you cannot call **delete** to destroy them. Readers familiar with OLE will recognize that the connector objects have internal reference counts that track the number of clients using them. Often you are not the only user of your own connectors. For example, when a server creates a *TOcRemView* to paint an object in a container's window, the container becomes a client of the same object. The server must not destroy the view object until the container is through with it, otherwise OLE could end up attempting to address functions that no longer exist in memory.

The Component Object Model decrees that an object must maintain an internal reference count. When an object provides anyone a pointer to one of its interfaces, the object also increments its own reference count. When the client finishes with the pointer, it calls *Release* and the object decrements its reference count. As long as the count is greater than zero, the object must not be destroyed. When the count reaches 0, the object destroys itself.

ObjectComponents shields you from the details of reference counting. You never have to increment or decrement a reference count. You cannot delete COM objects, however, because the **delete** command pays no attention to the reference count. Instead, call the connector's *ReleaseObject* method.

**Figure 18.11** How the ObjectComponents connector objects are related




Nonvirtual inheritance Virtual inheritance

## Automation classes

Table 18.4 describes some of the classes that appear in automation programs.

**Table 18.4** Some ObjectComponents classes used for automation

| Class | Description |
|---|---|
| TAutoBase | Simplifies clean-up chores when an automated object is destroyed. Make it the base class for your automated classes if you want that help. |
| TAutoProxy | The base class for an automation controller's proxy objects. Controllers create C++ proxy objects to represent the OLE objects they want to manipulate. The proxy objects become connected to OLE when they derive from *TAutoProxy*. |
| TOleAllocator | Initializes the OLE libraries and, optionally, passes OLE a custom memory allocator for managing any memory the system allocates on the program's behalf. |
| TRegistrar | Records application information in the system registration database and tells OLE when the application starts and stops. |

There are more automation classes than the table shows, but many of them are internal to the ObjectComponents implementation. Most of the work in automating an existing application is done with macros. Automating a class means writing two tables of macros, one in the class declaration and one in the class implementation. The macros describe the methods you choose to expose. Within the parent class they create nested

classes, one for each command. ObjectComponents knows how to make a nested object execute the method it exposes, and the nested class calls members of the parent class directly.

The connector objects that ObjectComponents creates to implement COM interfaces for an automation program are considered internal. ObjectComponents makes them for you when they are needed.

## ObjectComponents messages

When ObjectComponents needs to tell an application about signals and events that come from OLE, it sends a message through the normal Windows message queues. The message it sends is WM_OCEVENT. The value in the message's *wParam* identifies a particular event. Only applications that support linking and embedding receive WM_OCEVENT messages. (They are sent by the application's *TOcApp*, *TOcView*, and *TOcRemView* objects. Automation applications that don't support linking and embedding have no need for any of these objects.)

Simple ObjectWindows applications don't need to process any of the events because the new OLE classes have default event handlers that make reasonable responses for you. To modify the default behavior, add event handlers to your ObjectWindows program. For more information about handling events in ObjectWindows, see Chapter 4. If you are programming without ObjectWindows, handle WM_OCEVENT in your window procedure.

The events are divided into two groups. Those that concern the application as a whole are listed in Table 18.5. Those that call for a response from a particular document are addressed to the view window. They are listed in Table 18.6.

**Table 18.5** Application messages for TOcApp clients

| wParam value | Recipient | Description |
|---|---|---|
| OC_APPDIALOGHELP | Container | Asks the container to show Help for one of the standard OLE dialog boxes where the user has just clicked the Help button. |
| OC_APPBORDERSPACEREQ | Container | Asks the container whether it can give the server border space in its frame window. |
| OC_APPBORDERSPACESET | Container | Asks the container to rearrange its client area windows to make room for server tools. |
| OC_APPCREATECOMP | Server (and container acting as link source) | Asks the application to create a new component for embedding in another application. |
| OC_APPFRAMERECT | Container | Requests client area coordinates for the inner rectangle of the program's main window. |
| OC_APPINSMENUS | Container | Asks the container to merge its menu into the server's. |
| OC_APPMENUS | Container | Asks the container to install the merged menu bar. |
| OC_APPPROCESSMSG | Container | Asks the container to process accelerators and other messages from the server's message queue. |
| OC_APPRESTOREUI | Container | Tells the container to restore its normal menu, window titles, and borders because in-place editing has ended. |

**Table 18.5** Application messages for TOcApp clients (continued)

| wParam value | Recipient | Description |
|---|---|---|
| OC_APPSHUTDOWN | server | Tells the server when its last embedded object closes down. If the server has nothing else to do, it can terminate. |
| OC_APPSTATUSTEXT | Container | Passes text for the status bar from the server to the container. |

A *view* is the image of an object as it appears onscreen. When an OLE server gives an object to a container, the object contains data. The server also provides a view of the data so OLE can draw the object onscreen. Sometimes the word *view* also refers to the window where the container draws a compound document with all its embedded parts. Each object has its own small view, and the container has a single larger view of the whole document with all its embedded objects.

**Table 18.6** View messages for TOcView and TOcRemView clients

| wParam value | Recipient | Description |
|---|---|---|
| OC_VIEWATTACHWINDOW | Server | Asks server window to attach to its own frame window or container's window. |
| OC_VIEWBORDERSPACEREQ | Container | Asks whether server can have space for a tool bar within the view of an embedded object. |
| OC_VIEWBORDERSPACESET | Container | Asks container to rearrange its windows so the server can show its tool bar within an embedded object. |
| OC_VIEWCLIPDATA | Server | Asks server to provide clipboard data in a particular format. |
| OC_VIEWCLOSE | Server | Asks server to close its document. |
| OC_VIEWDRAG | Server | Asks server to provide visual feedback as the user drags its embedded object. |
| OC_VIEWDROP | Container | Tells container an object has been dropped on its window and asks it to create a *TOcPart*. |
| OC_VIEWGETPALETTE | Server | Asks server for the color palette it uses to draw an object. |
| OC_VIEWGETSCALE | Container | Asks container to give scaling information. |
| OC_VIEWGETSITERECT | Container | Asks container for the site rectangle that a part occupies. |
| OC_VIEWINSMENUS | Server | Asks server to insert its menus in a composite menu bar. |
| OC_VIEWLOADPART | Server | Asks server to load an embedded object stored in the container's data file. |
| OC_VIEWOPENDOC | Server | Asks server to open a document with the specified path. |
| OC_VIEWPAINT | Server | Asks server to draw or redraw an object at a particular position in a given device context. |
| OC_VIEWPARTINVALID | Container | Tells container that one of its embedded objects needs to be redrawn. |
| OC_VIEWPARTSIZE | Server | Asks server the initial size of its view in pixels. |
| OC_VIEWSAVEPART | Server | Asks server to write the data for an object into the container's file. |
| OC_VIEWSCROLL | Container | Asks container to scroll its view window. |

**Table 18.6** View messages for TOcView and TOcRemView clients (continued)

| wParam value | Recipient | Description |
|---|---|---|
| OC_VIEWSETSCALE | Server | Asks server to handle scaling. |
| OC_VIEWSETSITERECT | Container | Asks container to set the site rectangle. |
| OC_VIEWSHOWTOOLS | Server | Asks server to display its tool bar in container's window. |
| OC_VIEWTITLE | Container | Asks container for the caption in its frame window. |

Most of the events in Tables 18.5 and 18.6 are sent only to a server or to a container. A single application receives both kinds of messages if it chooses to support both container and server capabilities.

## Messages and windows

Because the view and part objects expect to send notification messages to a particular document, every ObjectComponents application is expected to create a new window for each open document. Document windows should not be frame windows; they should be client windows that exactly fill the client area of a parent frame window. In an SDI application, the parent is the application's main frame window. In an MDI application, the parent is an MDI child frame. ObjectWindows programs should use *TOleWindow* for client windows. Many ObjectWindows applications, including all those that use the Doc/View model, already possess client windows. For help implementing client windows with ObjectWindows, see "3. Setting up the client window" on page 319. To implement client windows in a C++ program, see "3. Creating a view window" on page 366.

## New ObjectWindows OLE classes

Another set of new classes integrates ObjectWindows with ObjectComponents. Internally, the new ObjectWindows classes use the the ObjectComponents classes to connect with OLE for you. Depending on the complexity of your ObjectWindows application, you might not need to interact directly with ObjectComponents at all. Table 18.7 briefly summarizes the most important new ObjectWindows classes.

**Table 18.7** New classes in ObjectWindows for OLE support

| New classes | Base classes | Description |
|---|---|---|
| TOleFrame | TDecoratedFrame | Provides OLE user interface support for the main window of an SDI application. |
| TOleMDIFrame | TMDIFrame and TOleFrame | Provides OLE user interface support for the main window of an MDI application. |
| TOleWindow | TWindow | Used as the client of a frame window, provides support for embedding objects in a compound document. |
| TStorageDocument | TDocument | Adds the ability to work with OLE's compound file structure. It is the natural class to choose for compound documents with embedded objects. |
| TOleDocument | TStorageDocument | Implements the Document half of an OLE-enabled Doc/View pair. |

**Table 18.7** New classes in ObjectWindows for OLE support (continued)

| New classes | Base classes | Description |
|---|---|---|
| TOleView | TOleWindow, TView | Implements the View half of a Doc/View pair. (For information about Doc/View pairs see Chapter 10.) |
| TOleFactory<><br>TOleDocViewFactory<><br>TOleAutoFactory<><br>TOleDocViewAutoFactory<><br>TAutoFactory<><br>TOcAutoFactory<> | TOleFactoryBase | Implements the function OLE calls when an application should create an object. |

The ObjectWindows OLE classes create ObjectComponents objects for you as needed. For example, whenever a container or a server creates a compound document, it also creates a a *TOcView* (or *TOcRemView*) object to implement the interfaces that tie a document to OLE. *TOleView::CreateOcView* does that for you. Furthermore, when the new *TOcView* object sends event messages to the view window, *TOleView* processes them for you with handlers like *EvOcViewSavePart* and *EvOcViewInsMenus*. The default event handlers manage much of the OLE user interface for you.

## Exception handling in ObjectComponents

ObjectWindows 2.5 modifies the hierarchy of exception classes. *TXBase* is the new base class for all exception classes. *TXOwl* derives from it, as do the new exception classes summarized in Table 18.8.

**Table 18.8** ObjectComponents exception classes

| Class | Purpose |
|---|---|
| TXAuto | Exceptions that occur during automation |
| TXObjComp | Exceptions that occur during ObjectComponents linking and embedding operations |
| TXOle | Exceptions that occur while processing OLE API commands |
| TXRegistry | Exceptions that occur while using the system registration database |

Because the exception classes all derive from *TXBase*, a general-purpose **catch** statement often takes a *TXBase&* as a parameter. The **catch** statement in the following example receives any exception thrown by ObjectWindows or ObjectComponents:

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    Registrar = new TOcRegistrar(AppReg, TOleFactory<TMyApp>(),
                                 TApplication::GetCmdLine());
    return Registrar->Run();
  }
  catch (TXBase& x) {
    ::MessageBox(0, x.why().c_str(), "Exception", MB_OK);
  }
```

```
    return -1;
  }
```

## TXOle and OLE error codes

Most of the OLE API functions pass back a return value of type HRESULT (or the nearly identical SCODE). The return value indicates whether the call was successful, and it can also encode other status information. When a public member function of an ObjectComponents class results in a call to an OLE interface and the interface call fails, then ObjectComponents turns the OLE return result into a C++ exception object of type *TXOle*. This allows you to handle OLE error codes via the standard C++ **try** and **catch** constructs.

The *TXOle* class defines a variable, *Stat*, which holds the return value passed back from from a failed OLE API call. Therefore, a **catch** statement taking a *TXOle&* as a parameter has access to the OLE error code. The following code shows an example of a routine where the error value is simply returned back to the caller. This is useful if the function is called from an application that cannot handle C++ exceptions.

```
HRESULT
TMyAppDescriptor::CheckTypeLib(TLangId lang, const char far* file)
{
  HRESULT stat = HR_NOERROR;

  // Create OCF classes and invoke OCF methods to perform operation
  try {
    TOleCreateList typeList(new TTypeLibrary(*this, lang), file);
    ⋮
  }

  catch(TXOle& x) {                   // Catch OLE exception
    stat = ResultFromScode(x.Stat);   // Create HRESULT from SCODE
  }
  return stat;                        // Return OLE error code
}
```

The previous example uses the *ResultFromScode* macro to cast an SCODE to an HRESULT. The OLE headers define various other macros that allow you to break down, assemble, and convert the various components of the value returned from an OLE API call. For more information, search for the topic "Error Handling Functions and Macros" in OLE.HLP.

If ObjectComponents catches a *TXOle* exception internally, it displays a dialog box showing the OLE return code. If the OLE_ERR.DLL library is present, ObjectComponents attempts to translate the error code into a string for the dialog box. Otherwise it displays just the numerical code.

OLE documents the codes only in the header files where they are defined. To make what information there is more accessible, the DOCS/OLE_ERR.TXT file extracts information from that header and presents the codes in numerical order. Also, the source code for the error message DLL is in SOURCE/OCTOOLS/OLE_ERR.

## Building an ObjectComponents application

All ObjectComponents applications require exception handling and RTTI. Do not set any compiler options that disable these features.

Linking and embedding applications must use the large memory model. Automation applications can use the medium model as well (and they run faster in medium model).

The integrated development environment (IDE) sets the appropriate compiler and linker options for you automatically when you select OCF in the TargetExpert.

To build any ObjectComponents program from the command line, create a short makefile that includes the OWLOCFMK.GEN found in the EXAMPLES subdirectory. If your application does not use ObjectWindows, include the OCFMAKE.GEN instead. Here, for example, is the makefile that builds the AutoCalc sample program:

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj autocalc.obj
HLP = MYPROGRAM
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJRES hold the name of the file to build and the names of the object files to build it from. HLP is optional. Use it if your project includes an online Help file. Finally, your makefile should include OWLOCFMK.GEN or OCFMAKE.GEN.

Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=l
```

MAKE, using instructions in the included file, will build a new makefile tailored to your project. The new makefile is called WIN16L*xx*.MAK. The final two digits of the name tell whether the makefile uses diagnostic or debugging versions of the libraries. *01* indicates a debugging version, *10* a diagnostic version, and *11* means both kinds of information are included. The same command also then runs the new makefile and builds the program. If you change the command to define MODEL as *d*, the new makefile is WIN16D*xx*.MAK and it builds the program as a DLL.

For more information about how to use OCFMAKE.GEN and OWLOCFMK.GEN, read the instructions at the beginning of MAKEFILE.GEN, found in the same directory.

Table 18.9 shows the libraries an ObjectComponents program links with.

**Table 18.9** Libraries for building ObjectComponents programs

| Medium model | Large model | DLL libraries | Description |
|---|---|---|---|
| OCFWM.LIB | OCFWL.LIB | OCFWI.LIB | ObjectComponents |
| OWLWM.LIB | OWLWL.LIB | OWLWI.LIB | ObjectWindows |
| BIDSM.LIB | BIDSL.LIB | BIDSI.LIB | Class libraries |
| OLE2W16.LIB | OLE2W16.LIB | OLE2W16.LIB | OLE system DLLs |
| IMPORT.LIB | IMPORT.LIB | IMPORT.LIB | Windows system DLLs |
| MATHWM.LIB | MATHWL.LIB | | Math support |
| CWM.LIB | CWL.LIB | CRTLDLL.LIB | C run-time libraries |

The ObjectComponents library must be linked first, before the ObjectWindows library.

## Distributing files with your application

When you distribute your application, you need to distribute along with it some libraries that ObjectComponents requires. Your installation program should install the files for the user, being careful not to replace any more current versions the user might already have.

The following files are part of OLE 2 and should be distributed with any 16-bit OLE application, whether it uses ObjectComponents or not.

| | | |
|---|---|---|
| compobj.dll | ole2conv.dll | ole2disp.dll |
| ole2.dll | ole2nls.dll | ole2prox.dll |
| storage.dll | typelib.dll | stdole.tlb |
| ole2.reg | | |

All these files belong in the user's WINDOWS/SYSTEM directory. Microsoft requires that if you distribute any of the files, you must distribute all of them. Call RegEdit to merge OLE2.REG with the user's registration database. (The RegEdit registration editor comes with Windows.) Double-clicking OLE2.REG in the File Manager accomplishes the same thing.

Any program that uses ObjectComponents should also distribute BOCOLE.DLL.

In addition, if your program uses the DLL version of OWL, of the container class libraries, or of the run-time library, you should distribute those as well.

# How ObjectComponents works

The information in this section is not essential for using ObjectComponents, only for understanding what goes on behind the scenes when you create ObjectComponents connector objects.

The essential function of ObjectComponents is to connect you with OLE. ObjectComponents is an intermediate layer standing between OLE on one side and your C++ code on the other.

## How ObjectComponents talks to OLE

Fundamentally, all OLE interaction of any sort requires the implementation of standard OLE interfaces, such as *IUnknown* and *IDispatch*, as defined by the Component Object Model (COM).

An *interface* is just a set of related function prototypes forming a pure base class. Every OLE object that implements the same interface can choose to implement the prescribed functions in its own way. All that matters is that the interface functions always accept the same parameters and always produce the same results. This makes it possible for any OLE object to call any standard function in any other OLE object that supports the interface.

Every OLE object must implement the *IUnknown* interface. One of the three functions in the *IUnknown* interface is *QueryInterface*. This common function implemented on all OLE objects lets you ask whether the object supports other interfaces that you want to use, such as automation interfaces or data transfer interfaces. This makes it possible for any OLE object to determine at run time what any other OLE object can do.

OLE defines a large number of standard interfaces that are notoriously tedious to implement. Borland's BOCOLE support library defines an alternate set of custom COM interfaces that collectively provide an alternative interface to OLE programming, one conceived at a higher level of abstraction. Client objects of the support library must still implement *IUnknown*, as all COM objects must, but instead of other standard OLE interfaces such as *IDataObject* and *IMoniker*, they implement interfaces defined by BOCOLE. The support library acts as an agent translating commands received through its custom interfaces into standard OLE. All the custom interfaces commands are carried out for you using standard OLE interfaces.

The custom interfaces in the BOCOLE support library have names like *IBContainer* and *IBDocument*. You'll see them used if you look in the ObjectComponents source code. Because the support library is an internal tool and subject to change, its interfaces are not documented. The complete library source code, however, comes with Borland C++, so you can refer to it if you need to track the OLE interactions minutely. You can also modify and rebuild the support library, just as you can the ObjectWindows Library, if that suits your purposes.

## How ObjectComponents talks to you

Some of the ObjectComponents classes define COM objects. These objects derive from *TUnknown*, an ObjectComponents base class that implements the *IUnknown* interface and handles details of aggregation (a way of combining several objects into a single functional unit). They also mix in other base classes that implement interfaces from the BOCOLE support library.

The ObjectComponents objects that implement COM interfaces are called *connector objects*, because they connect your application to OLE. *TOcPart*, for example, is the connector object that implements the interfaces a container must support for each OLE object (part) that is placed in its document. To embed an object in your document, you take information ObjectComponents gets from the Clipboard, a drop message, or the Insert Object dialog box, and you pass the information to the *TOcPart* constructor. Among other things, the constructor (indirectly) calls a BOCOLE function to create an embedded OLE object. *TOcPart* holds the pointer to that object, queries it for interfaces, and stores the coordinates of the site where the part should be drawn. When you want the part to do something, you call *TOcPart* methods such as *Activate* and *Save*.

## Linking and embedding connections

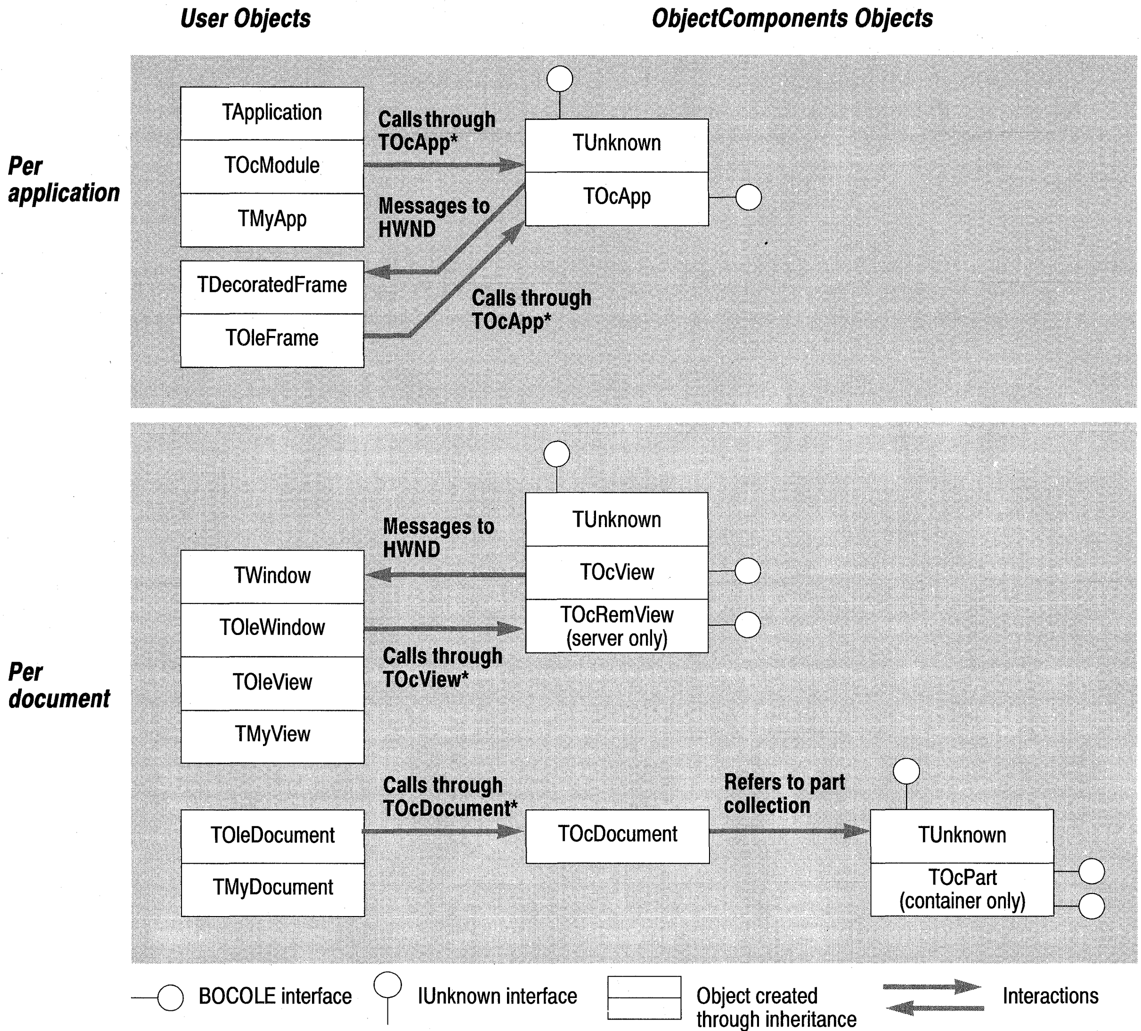
A linking and embedding application always creates a *TOcApp* object (usually it is created for you). *TOcApp* is a connector object that implements interfaces every linking and embedding application needs. Another connector object that all linking and embedding applications create is the view object, either *TOcView* for a container or *TOcRemView* for a server. You create one view object for each document you open. A

view object is associated with the window where the document is drawn. The only other connector object used for linking and embedding is *TOcPart*, which containers create for each object deposited in their documents.

Of course communication through a connector object is not just one way. When you call methods on a connector object, the object calls through to OLE, but sometimes OLE needs to call you. For example, if when user chooses Insert Object and asks for an object from a server, OLE must invoke the server and ask it to create an object. The connector objects cannot, of course, call your functions the same way you can call theirs because they don't know anything about your code. When a connector object needs to communicate a request or a notification from OLE to you, it sends WM_OCEVENT message to one of your windows. *TOcApp* sends its messages to your frame window. The view and part objects send messages to the client window where you draw your document.

Communication from you to OLE happens through function calls to connector objects. Communication from OLE to you happens through messages from connector objects to your windows. Figure 18.12 diagrams these interactions.

Figure 18.12 How objects in your application interact with ObjectComponents




The objects on the left side are instances of the ObjectWindows classes you normally create: an application, a frame window, a document, and a view. In applications that do not use the Doc/View model or do not use ObjectWindows, different classes fulfill the same functions. You always have a frame window and a document window, for example. The flow of interaction is the same in every ObjectComponents application.

The objects on the right side are the helpers from ObjectComponents that connect corresponding parts of your application to OLE.

The initial wiring between you and ObjectComponents is established the first time the registrar object calls your factory callback function. The *TOcApp* object is bound to a window in *TOleFrame::SetupWindow*, or in the WM_CREATE handler of your main window.

## Automation connections

Applications that support automation but not linking and embedding use a different set of objects. The central function of the automation layer in OLE is to pass arguments from the controller to the server, an operation with no user interface. The COM interfaces for automation are buried deeper in the implementation of ObjectComponents than the linking and embedding interfaces.

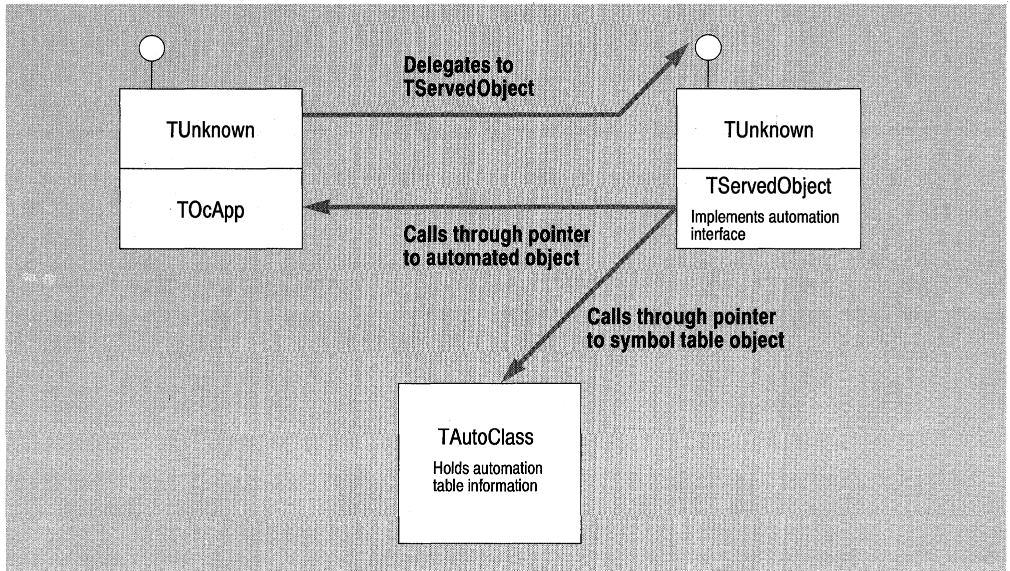
To support automation, ObjectComponents must identify exposed commands and arguments, attach type information to them, transfer values to and from the stack of VARIANT unions that OLE uses to pass values, and invoke your C++ functions when a controller sends a command. Once you set up the tables that describe what you want to expose, there is little in the automation process to customize or override. You never directly create or manipulate the connector objects for automation; ObjectComponents does it for you.

Advanced users who enjoy reading source code might like to know that *TServedObject* is the class that implements *IDispatch* and *ITypeInfo*, that *TTypeLibrary* implements *ITypeLib*, and that *TAutoIterator* implements *IEnumVARIANT*. Of these, only *TAutoIterator* is exposed as a public part of ObjectComponents. The others are considered internal implementation.

To automate a class, ObjectComponents asks you to build two descriptive tables from macros. A declaration table goes with the class declaration and declares which members are accessible to OLE. A definition table goes with the class implementation and assigns public names for controllers to use when invoking your functions. The automation macros also create nested classes within the automated parent, one for each exposed function or data member. The nested classes have an *Invoke* method that calls your function. Because the nested classes are friends of the surrounding class, they have direct access to it through normal C++ mechanisms.

*TServedObject* is the connector that receives *IDispatch* commands from OLE and translates them into the appropriate *Invoke* calls. *TServedObject* finds the information it needs to do this in an object of type *TAutoClass*, which holds the symbol information from the automation tables. *TServedObject* receives dispatch IDs, looks them up in *TAutoClass*, uses the information it finds to extract arguments from the stack of VARIANT unions passed by OLE. Finally it calls *Invoke* on the appropriate nested command object. Figure 18.13 diagrams the interaction of *TServedObject* with *TAutoClass* and your automated class.

**Figure 18.13** How TServedObject connects an automated class to OLE




# ObjectComponents Programming Tools

The most powerful tool in Borland C++ to help you with ObjectComponents programming is AppExpert. AppExpert generates a complete basic application according to your specification. It fully supports both linking and embedding and automation. Use it to create containers, servers, and automation servers. ClassExpert helps you modify the generated code to make it do what you need.

The TargetExpert in the integrated development environment (IDE) also supports ObjectComponents. Click the option for OCF and it automatically sets the right build options.

## Utility programs

Borland C++ 4.5 comes with some new utility programs that simplify common OLE programming chores. Some of them solve problems that other chapters explain in more detail.

**AutoGen**: Generates proxy classes for an automation controller. Scans the type library of an automated application and writes the source code for classes a controller uses to send commands automation commands.

**DllRun**: Launches a DLL server in executable mode. Any DLL server written with ObjectComponents can also run as a standalone application if you invoke it with DllRun. Running in executable mode sometimes makes it easier to debug the DLL. It also makes it possible to distribute a single program that your users can run either as an in-process server or as an independent application.

**GuidGen**: Generates globally unique identifiers for use in registering applications. Every server must have an absolutely unique ID. Containers need them in order to be link sources.

**MacroGen**: Generates automation macros for exposing functions with any number of arguments. The ObjectComponents headers declare versions of the macros for functions with up to four arguments. MacroGen saves you from having to revise the macros by hand to accommodate more arguments.

**Register**: Registers or unregisters any ObjectComponents EXE or DLL. Usually the applications register themselves if necessary when they run, or in response to command-line switches. Developers, however, sometimes need to register and unregister different versions of an application over and over. Register is especially useful for DLLs because you can't pass command-line switches to a DLL.

**WinRun**: A background program that makes it possible to launch Windows programs from the command line prompt in a DOS box. WinRun makes it possible to run GUI programs (such as Register) from a make file.

The source code for all the utilities but WINRUN is in the OCTOOLS directory.

You might find it helpful to install these tools in the integrated development environment (IDE). For more information, open the EXAMPLES\IDE\IDEHOOK\IDEHOOK.IDE file and read the instructions in OLETOOLS.CPP.

# Where do I look for information?

You can find information about programming with ObjectComponents in this book, in other books, in the online Help, and in the directories of sample programs.

Throughout the documentation, *OLE* refers to OLE 2.0 unless version 1 is indicated explicitly.

## Books

The chapters that follow describe how to build programs that perform all these functions.

**Table 18.10** Descriptions of the ObjectComponents chapters in this book

| Chapter Title | Topic |
|---|---|
| Support for OLE in Borland C++ | Overview of ObjectComponents |
| Creating an OLE container | How to build an application that receives OLE objects in its documents |
| Creating an OLE server | How to build an application that creates OLE objects for containers to use |
| Automating an application | How to build an application that other programs can control |
| Creating an automation controller | How to build an application that controls other applications |

For complete reference material covering all the new OLE-related classes and macros in ObjectComponents andObjectWindows, see the *ObjectWindows Reference Guide*.

The ObjectComponents material in this book and in the *ObjectWindows Reference Guide* is also in the online Help for Borland C++.

The *ObjectWindows Tutorial* develops a sample application from scratch. The later steps use add OLE container, server, and automation capabilities.

## Online Help

In addition, Borland C++ includes three online Help files covering the OLE API. For the most part, ObjectComponents makes knowledge of OLE interfaces unnecessary, but if you want to understand more about how ObjectComponents works, or if your application requires advanced programming at the OLE interface level, then you might find these files useful.

**Table 18.11** Online Help files with information about ObjectComponents and OLE

| Help file | Topic |
|---|---|
| OCF.HLP | ObjectComponents chapters from the *ObjectWindows Programmer's Guide* and the *ObjectWindows Reference Guide* |
| OWL.HLP | Reference material for new OLE-enabled classes in ObjectWindows |
| OLE.HLP | OLE system overviews and reference |

## Example programs

One of the best ways to learn about programming is to study working code. AppExpert is a good place to start. Use it to generate the code for servers, containers, automation servers, and DLL servers. In addition, Borland C++ comes with a variety of sample programs that show off ObjectComponents. Some of them are described in this list.

EXAMPLES/OCF: ObjectComponents without ObjectWindows

- **AutoCalc**: An automation server; draws a calculator onscreen and lets a controller click the buttons
- **CallCalc**: An automation controller to manipulate the calculator in AutoCalc
- **CppOcf**: Three-step linking and embedding tutorial that starts with a simple C++ program, turns it into a container, and then into a server
- **Localize**: Pulls translated strings from XLAT resources to reflect language settings
- **RegTest**: Registers, validates the registration, and unregisters an ObjectComponents application

EXAMPLES/OWL/TUTORIAL: ObjectWindows tutorial examples

- **OwlOcf**: Three-step linking and embedding tutorial that starts with a simple ObjectComponents program, turns it into a container, and then into a server.
- **Step14 – Step17**: The final steps of the tutorial application described in *ObjectWindows Tutorial*; shows how to be a linking and embedding container or

server, how to be an automation server or controller, and how to support both automation and linking and embedding at the same time

EXAMPLES/OWL/OCF: ObjectComponents with ObjectWindows

- **MdiOle**: A multidocument interface application with container capabilities
- **SdiOle**: A single document interface application with container capabilities
- **Tic Tac Toe**: A linking and embedding server

SOURCE/OCTOOLS: source code for programming utilities

- **AutoGen**: Scans a type library and generates proxy classes for an automation controller
- **DllRun**: Runs a DLL server in executable mode
- **GuidGen**: Generates globally unique identifiers (GUIDs)
- **Register**: Registers or unregisters a server (EXE or DLL)

## Glossary of OLE terms

The definitions in this list explain common terms in OLE programming. Read it for an introduction to important programming topics, or refer to it for clarification as you read other ObjectComponents chapters.

The definitions of advanced concepts assume you already know something about OLE and its standard interfaces. For more information about OLE, refer to the three OLE online Help files.

- **Activate**: the user *activates* a linked or embedded object by double-clicking it. Activating an object causes the server to execute the object's primary verb. For document-style objects, the primary verb is generally initiates an editing session, either in-place or open. For other objects, such as movies and sounds, the primary verb is usually Play. Activating is not the same as selecting; see the entry for *Select*.
- **Aggregation**: a way of combining several OLE objects to make them function as a single bigger object. Objects are aggregated at run time. You can aggregate with objects that you did not design. An object aggregates to delegate commands or to inherit and override the functionality of other objects.

  Aggregation is an advanced programming technique. In order for aggregated objects to act as a unit, all the aggregated objects must delegate any *QueryInterface* call they receive to the primary object, usually called the outer object. The outer object begins an aggregation by passing its own *IUnknown* pointer. The second object remembers the outer *IUnknown* pointer and routes all requests for an interface to the outer object. If the outer object does not support a requested interface, it forwards the request to the first in what might be a chain of aggregated objects. A client can reach all the interfaces supported by any of the auxiliary objects through the *IUnknown* of the outer object.
- **Automated object** or **application**: an OLE object that publishes commands other applications can send it. An automation server creates automated objects. The

automated object can be the application itself or something that the application creates.

- **Automation**: the ability of an application to define a set of commands for other applications to invoke.
- **Automation controller**: an application that invokes commands to control automated objects or applications. A controller is sometimes called an *automation client*.
- **Automation server**: an application that exposes some of its own internal function calls as a set of commands that other programs can invoke. An *automation object* is what the server creates for other programs to control.
- **BOCOLE support library**: a DLL of OLE implementation utility interfaces that ObjectComponents calls internally. The support library implements a number of custom OLE interfaces designed by Borland. The BOCOLE.DLL file should be distributed with any ObjectComponents program. Its custom interfaces are considered internal and so are not documented. The source code for the BOCOLE support library, however, is included with Borland C++.
- **COM object**: An object whose architecture conforms to the Component Object Model, a Microsoft specification that forms the basis of the OLE system. Briefly stated, the characteristics of COM objects are
  - They communicate through predefined interfaces.
  - They all support the *IUnknown* interface, and *IUnknown* includes the *QueryInterface* method for getting other optional interfaces.
  - They keep a reference count of their clients and delete themselves if the count reaches zero.

  Only COM objects can communicate with OLE. Some of the classes in ObjectComponents are COM objects (see *Connector object*). ObjectComponents shields you from the details of interface methods, interface pointers, and reference counters. It connects you to OLE using familiar C++ and Windows programming models such as inheritance and messages.
- **Compound document**: a document that contains OLE objects brought in from other applications. A compound document might contain pieces of information from a spreadsheet, a database, and a word processor, all in one document that the user loads or saves with a single command. The objects from other applications are either linked or embedded in the container's document.
- **Compound file**: a single disk file that the operating system divides into independent compartments called *storages*. In effect, each storage has its own file I/O pointer so you can read, write, rewrite, and erase data in any one storage without needing to maintain offsets to other storages in the same file. Compound files are useful for storing compound documents because you can create a new storage for each linked or embedded object. OLE extends the file system by implementing interfaces to support compound files.
- **Connector object**: an ObjectComponents class that communicates with OLE for you. Connector objects connect parts of your application to OLE. *TOcApp*, for example, performs OLE functions for the application. *TOcView* performs OLE functions for one view of a document. *TOcPart* performs OLE functions for a linked or embedded

object. The connector objects are partners that work together with corresponding parts of your application. You call their methods and they send you messages. Connectors are Component Object Model objects and implement COM interfaces. (Not all ObjectComponents classes are connectors.)

- **Container**: an application that permits OLE to embed or link objects from other applications into its own documents. Containers are also called *clients* of the servers that give them objects.
- **DLL server**: a server whose code is in a dynamic-link library rather than an executable file. The advantage of a DLL server is speed. When OLE invokes an .EXE server to support an embedded object, it has to create a a separate process and marshall data to pass it between the two applications. A DLL, on the other hand, is part of the same system task as its client, so OLE calls from a container to a DLL server run much more quickly. See "Making a DLL server" on page 374.
- **Document**: this word has two different meanings for programmers. First, a document is a set of data that an application loads in response to File | Open. A document can be a spreadsheet, or a bitmap, or a letter, or any other set of data that an application treats as a whole.

  Sometimes it is useful to distinguish between the data in a document and the appearance of the data onscreen. A spreadsheet, for example, might be able to display a single set of data as either a table of numbers or a chart. One document can be displayed different ways. In such cases, *document* refers only to the data, and each possible representation of the document is called a *view.*

  ObjectWindows programmers are familiar with an application architecture called the *Doc/View model* that separates the code for managing document data from the code for displaying the data. ObjectComponents also has a document class and view classes, but they are not part of the ObjectWindows Doc/View model. The document class keeps track of the objects embedded in a document and the view classes draw the objects onscreen.
- **Embedded object**: data from a server application deposited by OLE in a container's document. OLE lets the user paste, drag, or insert objects into a container. If during these actions the user chooses to create an *embedded* object, then all the data in the object is copied to the container's document. When the user loads or saves the document, the data for the embedded object is written to the file along with the container's own native data.

  Contrast embedded objects with *linked* objects, where the the data for the OLE object is stored in another application and the container receives only a reference to the object's file.
- **EXE server**: aserver application compiled and linked into an executable file. A server can also be built as a library; see *DLL server*.
- **GUID**: globally unique identifier, a 16-byte value. OLE uses GUIDs to identify applications, the objects they produce, and the interfaces that objects implement. For linking and embedding, OLE needs GUIDs to match embedded objects to their servers even after the user transfers a compound document from system to system. If two servers had the same ID, OLE might accidentally invoke the wrong one. Each

server and object type must have an absolutely unique ID. Tools such as GUIDGEN create the ID for you. For more information, see the *clsid* entry in the *ObjectWindows Reference Guide*.

- **IDispatch interface**: the OLE interface that all automated objects implement. With the four methods of the *IDispatch* interface, you can ask any automated object for information about its automated commands, look up the identifiers for particular commands, or invoke any command. For more information, see the OLE.HLP Help file.

- **In-place editing:** editing an OLE object in the container's window. During in-place editing, the container lets the server display its own menus and tool bars in the container's window. The purpose of in-place editing is to let the user edit any object in a document without leaving the document's window. In-place editing is illustrated in Figure 18.3 on page 270. Contrast *Open editing*.

- **In-process server:** same as DLL server.

- **Interface**: a set of function prototypes, usually declared as an abstract base class. OLE objects are able to communicate with each other because they implement standard interfaces, sets of functions that the system defines. The system defines only what functions an interface contains; it does not implement the functions. Each object implements the functions for itself. The interfaces are defined in the OLE system headers such as compobj.h and ole2.h. The OLE system communicates with applications and objects by calling the functions it assumes each one has implemented. For more about the OLE interface model, see the entry for *Component Object Model* (COM). For examples of standard OLE interfaces, see *IDispatch* and *IUnknown*.

  Besides the standard interfaces, an object can define and implement its own *custom* interfaces. Of course the system can't call functions from custom interfaces because it doesn't know they exist, but other applications that know about the custom interface can use it. Internally, ObjectComponents works through a set of Borland custom interfaces. See *BOCOLE support library*.

  ObjectComponents shields you from having to understand or implement particular interfaces. Advanced users who want to manipulate interfaces directly or mix in their own custom interfaces are free to do so.

- **IUnknown interface**: the root interface that all OLE objects and interfaces must implement. With the three methods of the *IUnknown* interface, you can ask any object for a pointer to another interface it might also support, and you can adjust the object's reference count. For more information, see the OLE.HLP Help file.

- **Linked object**: an object that appears in a container document but whose data really resides in another file. When dragging or pasting an object into a container, the user can choose to create a *link* to the object instead of embedding it. The container does not receive or store the linked object's data in its own document. Instead, it receives only a string identifying the location of the actual data, which can be in a file.

  Several containers can link to the same object. In that case, all the containers receive the same string pointing to the same object. If the data in the original object changes, then the changes are reflected automatically in all the documents that link to it. If the

user *embeds* one object in several containers, then each container has its own copy of the object's data and changes in one copy do not affect the other copies.

- **Link source**: the document that a link refers to, the source for the data in a linked object. Usually the link source is a server document, but it is not uncommon for containers to export link source data so that other applications can link to objects embedded or linked in the container's document. For information on becoming a link source, see the entry for REGFORMAT in the *ObjectWindows Reference Guide*.
- **Localization:** adapting an application to display strings in the user's language, whatever that might be. OLE servers need to speak the language of their client programs. If an automation server is marketed in several countries, it needs to recognize commands sent in each different language. A linking and embedding server registers strings that describe its objects to the user, and those too should be available in multiple languages in order to accommodate whatever language the user might request. ObjectComponents lets you place translations for all your strings in your resource file as XLAT resources. ObjectComponents chooses the right string at the right time.
- **ObjectComponents Framework**: a set of C++ classes from Borland International that encapsulate linking and embedding functions as well as automation functions. Internally the ObjectComponents classes implement standard and custom OLE interfaces. With ObjectComponents you write for OLE using familiar programming models such as inheritance and window messages instead of implementing COM interfaces.
- **ObjectWindows Library**: a set of C++ classes from Borland International that encapsulate standard Windows programming functions such as managing windows and dialog boxes. The current version of ObjectWindows introduces some new classes, such as *TOleWindow* and *TOleView*, that use ObjectComponents classes to acquire OLE capabilities. The new classes make it very easy to add OLE support to existing ObjectWindows applications.
- **OLE**: object linking and embedding, an extension to the Windows system. (In newer versions of Windows, OLE is an integral part of the system , not an extension.) The new commands that OLE implements and the interfaces it defines add many new features to the system, including linking and embedding, automation, and compound file I/O.
- **OLE interface**: see *Interface*.
- **Open editing:** editing an OLE object in the server's own window. Open editing happens when the user executes the Open verb. During open editing, the server's window opens up in front of the container's window. When the user finishes editing the object, the server window disappears and the modifications become visible back in the container window. Open editing is illustrated in Figure 18.9 on page 275. Contrast *In-place editing*.
- **Part**: an object linked or embedded in a compound document. An ObjectComponents container creates an object of class *TOcPart* to represent each object linked or embedded in its document.

*Part* is the container's word for an object created by a server. In the server's code, the same object is created as a normal server document. ObjectComponents presents the document to OLE as an OLE object. The container, when it receives the OLE object, creates a *TOcPart*. When the part needs to be painted, the part object communicates through OLE with the server's view object.

- **Reference counting**: a way of remembering how many clients an object has. Every section of code that requires the object to exist calls the object's *AddRef* method to increment the reference count. When the client code is done, it calls the object's *Release* method to decrement the reference count. If a *Release* call causes the count to reach 0, then the object is allowed to destroy itself.

  Every OLE object has *AddRef* and *Release* methods because they are part of the *IUnknown* interface. Knowing who is a client and when to call *AddRef* or *Release* is sometimes complicated. ObjectComponents manages reference counting for you. Only advanced users will find any need to call *AddRef* or *Release* directly.

- **Registrar object**: an object of type *TRegistrar* or *TOcRegistrar*. Every ObjectComponents application needs a registrar object. The registrar processes the application's command line, sets running mode flags, verifies the application's entries in the system registration database, and calls the application's factory function to launch the application.

- **Registration**: giving information about the application to the system. OLE programs perform two different kinds of registration. When an application is first installed, ObjectComponents writes information from the application's registration tables into the system registration database. This information is static and needs to be recorded only once. The registrar object performs this task.

  Subsequently whenever the user launches the application, ObjectComponents tells OLE that the application is running and it registers a factory for each type of document the application can produce. When the application ends, ObjectComponents unregisters the factories. The *TOcApp* or *TRegistrar* object performs this task.

- **Registration database:** see *System registration database*.

- **Registration table:** a table built with registration macros and containing information about an application or about the types of documents an application creates. The macros create a structure of type *TRegList*. The registrar object reads the registration structure and copies any necessary information to the system registration database.

- **Remote view**: the view of its own object a server draws in a container's window. When an ObjectComponents server is launched to manage an object linked or embedded in a container's document, the server creates a *TOcRemView* object and a *TOcDocument* object. The view object draws in the container's window. The document object loads and saves the object's data.

- **Select**: the user *selects* an object by clicking it once. The selected object does not become active and cannot be edited. Conventionally a container indicates that an object is selected by drawing a rectangle with grapples around the object. (*Grapples* are small handles for moving the rectangle.) The container might permit the user to select several objects at once to move or delete as a group, but usually only one object per child window can be active at a time.

- **Server**: an application that creates objects for other applications to use. In this documentation, *server* usually refers to either a linking and embedding server or an automation server. A linking and embedding server creates data objects that containers can paste, drop, or insert into their own compound documents. An automation server creates objects that other applications can manipulate by sending commands for the object to execute. (A single application can choose to create both kinds of objects. It is even possible to link and embed automated objects.)

- **System registration database**: a structured repository of information about applications installed on a particular computer. In 16-bit Windows, the database is kept in the REG.DAT file. In 32-bit Windows, the database is called the *system registry* and resides in private system files. Applications record their information during installation. The information includes identifiers for the application and its documents, descriptions of the application and its documents, the path to the application file, the default extension of the application's document files, and other details that help the OLE system associate servers with their objects.

- **Type library**: a file describing the commands an automation controller supports. Creating a type library is the standard way for an automation server to publish the programming interface it implements. The type library tells what objects the server creates and describes the objects' properties and methods. Type information is read by compilers and interpreters that process automation commands. Some applications also allow the user to browse the type information.

  Any ObjectComponents automation server generates a type library if you invoke it with the -TypeLib command line switch. Type libraries conventionally use the .TLB or .OLB extension. An automation server registers the location of its type library during installation.

- **Verb**: a command that a linking and embedding server can execute with its objects. The server tells the container what verbs it supports and the container displays the verb strings on its own Edit menu. To execute a verb, the user selects an object and then chooses a verb from the menu. The container updates the verb menu each time the user selects a new object.

  The server can support any verbs it chooses. Most servers support the Edit and Open verbs for in-place or open editing. Depending on the kind of data it owns, a server might choose to offer other verbs such as Play and Rewind.

- **View**: the graphical representation of data. The term is used to distinguish between the way the data is painted and the data itself, usually called the *document*. A single word processor document, for example, might have three different views: a page layout view, a draft view without fancy fonts, and a print preview view.

In ObjectComponents, containers create views to draw their compound documents. Servers also create views to draw the objects they create. Both create a *TOcDocument* object to manage the data and a view object, either *TOcView* or *TOcRemView*, to draw the document.

In ObjectWindows, *Doc/View* refers to a particular application architecture supported by the framework that also treats data and its representation in separate classes.

Chapter

# 19

# Creating an OLE container

An OLE *container* is an application that can store in its own documents data objects taken from other applications. A container can link objects or embed them in its documents. A program that creates objects to be linked or embedded is called a *server*.

This chapter explains how to take existing programs and turn them into OLE containers. It describes the steps required for adapting three different kinds of programs:

- An ObjectWindows application that uses the Doc/View model
- An ObjectWindows application that does not use the Doc/View model
- A C++ application that does not use ObjectWindows

The first case turns out to be very simple. The last case, relying entirely on the ObjectComponents Framework, requires the most new code, but it is still substantially easier than programming directly to OLE.

## Turning a Doc/View application into an OLE container

Turning a Doc/View application into an OLE container requires only a few modifications. The following list describes the changes briefly. Subsequent sections give more detail for each one.

1 Connect your application, window, document, and view objects to OLE.

   - Derive your application class from *TOcModule* as well as *TApplication*.
   - Derive frame window, document, and view classes from new OLE-enabled classes.
   - Create a *TAppDictionary* object.

2 Register the application.

   - Using macros, build registration tables to describe your application.
   - Create a registrar object and call its *Run* method.

3 Support OLE commands.

- Set up your Edit menu and tool bar using the appropriate predefined identifiers to support standard OLE commands.
- Make your Open and Save commands read and write embedded objects in your compound documents.

4 Build the container application.

- Include new ObjectWindows OLE headers at the beginning of your source code.
- Compile the program using the large memory model. Link to the OLE and ObjectComponents libraries.

That's all you need to do. By following these steps, you can create an OLE container that supports all the following features:

- Linking
- OLE clipboard operations
- In-place editing
- Compound document storage
- Embedding
- Drag and drop operations
- Tool bar and menu merging

You also get standard OLE 2 user interface features, such as object verbs on the Edit menu, the Insert Object dialog box, and a pop-up menu that appears when the user right-clicks an embedded object.

ObjectComponents provides default behavior for all these common OLE features. Should you want to modify the default behavior, you can additionally choose to override the default event handlers for messages that ObjectComponents sends. For a list of the event messages, see Tables 18.5 and 18.6.

The code examples in this section are based on the STEP14.CPP and STEP14DV.CPP sample programs in EXAMPLES/OWL/TUTORIAL. Look there for a complete working program that incorporates all the prescribed steps.

# 1. Connecting objects to OLE

Your application, window, document, and view objects need to make use of new OLE-enabled classes. The constructor for the application object expects to receive an application dictionary object, so create that first.

## Deriving the application object from TOcModule

The application object of an ObjectComponents program needs to derive from *TOcModule* as well as *TApplication*. *TOcModule* coordinates some basic housekeeping chores related to registration and memory management. It also connects your application to OLE. More specifically, *TOcModule* manages the connector object that implements COM interfaces on behalf of an application.

If the declaration of your application object looks like this:

```
class TMyApp : public TApplication {
  public:
    TMyApp() : TApplication(){};
  :
};
```

Then change it to look like this:

```
class TMyApp : public TApplication, public TOcModule {
  public:
    TMyApp(): TApplication(::AppReg["appname"], ::Module, &::AppDictionary){};
  :
};
```

The constructor for the revised *TMyApp* class takes three parameters.

- A string naming the application

  *AppReg* is the application's registration table, shown later in "Building registration tables." The expression *::AppReg["appname"]* extracts a string that was registered as the application's name.

- A pointer to the application module.

  *Module* is a global variable of type *TModule** defined by ObjectWindows.

- The address of the application dictionary.

  *AppDictionary* is the application dictionary object explained in the previous section.

## Inheriting from OLE classes

ObjectWindows includes classes that let windows, documents, and views interact with the ObjectComponents classes. The ObjectWindows OLE classes include default implementations for most normal OLE operations. To adapt an existing ObjectWindows program to OLE, change its derived classes so they inherit from the OLE classes. Table 19.1 shows which OLE class replaces each of the non-OLE classes.

**Table 19.1** Non-OLE classes and the corresponding classes that add OLE support

| Non-OLE class | OLE class |
|---|---|
| TFrameWindow | TOleFrame |
| TMDIFrame | TOleMDIFrame |
| TDecoratedFrame | TOleFrame |
| TDecoratedMDIFrame | TOleMDIFrame |
| TWindow | TOleWindow |
| TDocument | TOleDocument |
| TView | TOleView |
| TFileDocument | TOleDocument |

The *TOleFrame* and *TOleMDIFrame* classes both derive from decorated window classes. The OLE 2 user interface requires containers to handle tool bars and status bars. Even if

the container has no decorations, servers might need to display their own in the container's window. The OLE window classes handle those negotiations for you.

Wherever your existing OWL program uses a non-OLE class, replace it with an OLE class, as shown here. Boldface type highlights the change.

**Before**

```
// pre-OLE declaration of a window class
class TMyFrame: public TFrameWindow    { /* declarations */ );
```

**After**

```
// new declaration of the same window class
class TMyFrame: public TOleFrame     { /* declarations */ );
```

**Note** If the implementation of your class makes direct calls to its base class, be sure to change the base class calls, as well. Response tables also refer to the base class and need to be updated.

### Creating an application dictionary

An *application dictionary* tracks information for the currently active process. It is particularly useful for DLLs. When several processes use a DLL concurrently, the DLL must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. For example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* corresponding to the currently active client process. If you convert an executable server to a DLL server, your application too must maintain a dictionary of the *TApplication* objects representing each of its container clients. If your DLL uses the DLL version of ObjectWindows, then your DLL needs its own dictionary and cannot use the one in ObjectWindows.

The DEFINE_APP_DICTIONARY macro provides a simple and unified way to create the application object for any application, whether it is a container or a server, a DLL or an EXE. Insert this statement with your other static variables:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

For any application linked to the static version of the DLL, the macro simply creates a reference to the application dictionary in ObjectWindows. For DLL servers using the DLL version of ObjectWindows, however, it creates an instance of the *TAppDictionary* class.

**Note** Name your dictionary object *AppDictionary* to take advantage of the factory templates such as *TOleDocViewFactory* (as explained in the section, "Creating a registrar object").

## 2. Registering a container

To register your application with OLE, create registration tables describing the application and the kinds of documents it creates. Create a registrar object to process the information in the tables.

## Building registration tables

OLE requires programs to identify themselves by registering unique identifiers and names. OLE also needs to know what Clipboard formats a program supports. Doc/View applications also register their document file extensions and document flags. To accommodate the many new items an application might need to register, in ObjectWindows 2.5 you use macros to build structures to hold the items. Then you can pass the structure to the object that needs the information. The advantage of this method lies in the structure's flexibility. It can hold as many or as few items as you need.

**Note** Previous versions of ObjectWindows passed some of the same information in parameters. Old code still works unchanged, but passing information in registration structures is the recommended method for all new applications.

A Doc/View OLE container fills one registration structure with information about the application and then creates another to describe each of its Doc/View pairs. The structure with application information is passed to the *TOcRegistrar* constructor, as you'll see in the next section. Document registration structures are passed to the document template constructor.

Here are the commands to register a typical container:

```
REGISTRATION_FORMAT_BUFFER(100)      // allow extra space for expanding macros

BEGIN_REGISTRATION(AppReg)           // information for the TOcRegistrar constructor
  REGDATA(clsid,      "{383882A1-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(appname,    "DrawPad Container")
END_REGISTRATION

BEGIN_REGISTRATION(DocReg)           // information for the document template
  REGDATA(progid,     "DrawPad.Document.14")
  REGDATA(description,"Drawing Pad (Step14--Container)")
  REGDATA(extension,  "p14")
  REGDATA(docfilter,  "*.p14")
  REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGFORMAT(0, ocrEmbedSource,  ocrContent,  ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent,  ocrMfPict|ocrStaticMed, ocrGet)
  REGFORMAT(2, ocrBitmap, ocrContent,  ocrGDI|ocrStaticMed, ocrGet)
  REGFORMAT(3, ocrDib, ocrContent,  ocrHGlobal|ocrStaticMed, ocrGet)
  REGFORMAT(4, ocrLinkSource, ocrContent,  ocrIStream, ocrGet)
END_REGISTRATION
```

The registration macros build structures of type *TRegList*. Each entry in a registration structure contains a key, such as *clsid* or *progid*, and a value assigned to the key. Internally ObjectComponents finds the values by searching for the keys. The order in which the keys appear does not matter.

Insert the registration macros after your declaration of the application dictionary. Since the value of the *clsid* key must be a unique number identifying your application, it is recommended that you generated a new value using the GUIDGEN.EXE utility. (The *ObjectWindows Reference Guide* entry for *clsid* explains other ways to generate an identifer.) Of course, modify the value of the *description* key to describe your container.

The example builds two structures, one named *AppReg* and one named *DocReg*. *AppReg* is an *application registration structure* and *DocReg* is a *document registration structure*. Both structures are built alike, but each contains a different set of keys and values. The keys in an application registration structure describe attributes of the application. A document registration structure describes the type of document an application can create. A document's attributes include the data formats that it can exchange with the clipboard, its file extensions, and its document type name.

The set of keys you place in a structure depends on what OLE capabilities you intend to support. The macros in the example show the minimum amount of information a container should provide.

Table 19.2 briefly describes all the registration keys that a container can use. It shows which are optional and which required as well as which belong in the application registration table and which in the document registration table.

**Table 19.2** Keys a container registers to support linking and embedding

| Key | In AppReg? | In DocReg? | Description |
|---|---|---|---|
| appname | Optional | No | A short name for the application |
| clsid | Yes | Optional | Globally unique identifier (GUID); generated automatically for the *DocReg* structure. |
| description | No | Yes | Descriptive string (up to 40 characters) |
| progid | No | Yes for a link source | Identifier for program or document type (unique string) |
| extension | No | Optional | Document file extension associated with server |
| docfilter | No | Yes | Wildcard file filter for File Open dialog box |
| docflags | No | Yes | Options for running the File Open dialog box |
| format*n* | No | Yes | A clipboard format the container supports |
| directory | No | Optional | Default directory for storing document files |
| permid | No | Optional | Name string without version information |
| permname | No | Optional | Descriptive string without version information |
| version | Optional | No | Major and minor version numbers (defaults to "1.0") |

The table shows what is required for container documents that let other containers link to their embedded objects. For documents that do not support linking to embedded objects, the container needs to register only *docflags* and *docfilter*.

If your container is also a linking and embedding server or an automation server, then you should also consult the server table on page 346 or the automation table on page 384. Register all the keys required in any tables that apply to your application.

For more information about registration tables, see "Understanding registration" on page 372. For more information about individual registration keys and the values they hold, see the *ObjectWindows Reference Guide*.

The values assigned to keys can be translated to accommodate system language settings. For more about localization, see the section "Registering localized entries" on page 373 and "Localizing symbol names" on page 397.

## Understanding registration macros

The first macro in the example, REGISTRATION_FORMAT_BUFFER, sets the size of a buffer needed temporarily as the macros that follow are expanded. For more about about determining the buffer size, see page 347.

The REGDATA, REGFORMAT, and REGDOCFLAGS macros place items in the registration structure. All the registration macros are documented in the *ObjectWindows Reference Guide*.

REGDATA's first parameter is a key and the second is a value to associate with the key. In the example, the *AppReg* structure begins by assigning a value to the key *clsid*. The *clsid* is a globally unique identifier (GUID) specifying the application. The application's *progid* is a text string that serves the same purpose. The *description* key briefly describes the application (*Drawing Pad* (*Step14—Container*)). Of these three keys, only the *description* value is visible to users. (Users also see the *progid* if the application is automated; see Chapter 21, "Automating an application.") The document structure registers its own *progid* and *description*. Although each document type also needs its own unique *clsid*, if you omit it ObjectComponents supplies it for you by incrementing the application's *clsid*.

REGFORMAT entries list the data formats that the container can place on the Clipboard. The first parameter sets a priority order for the formats you use. *0* marks the format that renders data with the best fidelity, and higher numbers indicate lower fidelity. The second parameter represents a data format. The other parameters tell what presentation aspect of the format you use, what medium you use to transfer the data, and whether you can supply and receive Clipboard data in that format. All the data formats you specify with REGFORMAT are registered with the Windows Clipboard for you.

Even a simple container is usually capable of placing OLE objects on the Clipboard. If the user selects a linked or embedded object from the container's document and wants to transfer it through the Clipboard to another container, then the first container needs to act as a server by supporting at least the *ocrEmbedSource* or *ocrLinkSource* formats. Any application that registers either of these formats must also register *ocrMetafilePict*. The usual case is to register the five formats shown in the example. ObjectComponents automatically handles OLE objects in any of the standard formats for you. All you have to do is register the ones you want to support.

To register user-defined formats, replace the data format parameter with a string naming your format.

```
REGFORMAT(3, "MyOwnFormat", ocrContent, ocrIStorage, ocrGet)
```

If you register any custom Clipboard formats, you must also provide OLE with strings to describe your format in dialog boxes. Call *AddUserFormatName*, a method on classes derived from *TOleFrame*, to supply the descriptions.

For more information, see REGFORMAT in the *ObjectWindows Reference Guide*.

REGDOCFLAGS adds to the registration structure an entry containing flags for a document template. The flags set options for running the File Open common dialog box.

After creating registration tables, you must pass them to the appropriate object constructors. The *AppReg* structure is passed to the *TOcRegistrar* constructor, as

described in "Creating a registrar object." In a Doc/View application, document registration tables are passed to the document template constructor.

```
DEFINE_DOC_TEMPLATE_CLASS(TMyOleDocument, TMyOleView, MyTemplate);
MyTemplate myTpl(DocReg);
```

A program that uses several document templates should create a different registration table for each template. Each registration table must start with the BEGIN_REGISTRATION macro and have a different name, for example *DocReg1* and *DocReg2*.

All the information that normally gets passed to a document template constructor can be placed in a registration structure using REGFORMAT, REGDOCFLAGS, and REGDATA. Previous versions of OWL passed the same information to the document template as a series of separate parameters. The old method is still supported for backward compatibility, but new programs, whether they use OLE or not, should use the registration macros to supply document template parameters.

## Creating a registrar object

Every ObjectComponents application needs a registrar object to manage its registration tasks. In a linking and embedding application, the registrar is an object of type *TOcRegistrar*. At the top of your source code file, declare a global variable holding a pointer to the registrar.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TPointer* template ensures that the *TOcRegistrar* instance is deleted when the program ends.

**Note** Name the variable *Registrar* to take advantage of the factory callback template used in the registrar's constructor.

The next step is to modify your *OwlMain* function to allocate a new *TOcRegistrar* object and initialize the global pointer *Registrar*. The *TOcRegistrar* constructor expects three parameters: the application's registration structure, the component's factory callback and the command line string that invoked that application.

- The registration structure you create with the registration macros.
- The factory callback you create with a template class.

  For a linking and embedding ObjectWindows application that uses Doc/View, the template class is called *TOleDocViewFactory*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.
- The command line string can come from the *GetCmdLine* method of *TApplication*.

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    // Create Registrar object
    Registrar = new TOcRegistrar(::AppReg, TOleDocViewFactory<TMyApp>(),
                                 TApplication::GetCmdLine());
```

```
    return Registrar->Run();
  }
  catch (xmsg& x) {
    ::MessageBox(0, x.why().c_str(), "Exception", MB_OK);
  }
  return -1;
}
```

After initializing the *Registrar* pointer, your OLE container application must invoke the *Run* method of the registrar instead of *TApplication::Run*. For OLE containers, the registrar's *Run* simply invokes the application object's *Run* to create the application's windows and process messages. However, using the registrar method makes your application OLE server-ready. The following code shows a sample *OwlMain* before and after the addition of a registrar object. Boldface type highlights the changes.

Before:

```
// Non-OLE OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TMyApp().Run();
}
```

After adding the registrar object:

```
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg,
         TOleDocViewFactory<TMyApp>(),
         TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

The last parameter of the *TOcRegistrar* constructor is the command line string that invokes the application. The registrar object processes the command line by searching for switches, such as -Embedding or -Automation, that OLE may have placed there. ObjectComponents takes whatever action the switches call for and then removes them. If for some reason you need to test the OLE switches, be sure to do it before constructing the registrar. If you have no use for the OLE switches, wait until after constructing the registrar before parsing the command line. For more information about command line switches, see "Processing the command line" on page 349.

## 3. Supporting OLE commands

A container needs to place some standard OLE commands on its Edit menu. ObjectWindows implements the commands for you. A container also needs to let ObjectComponents read and write any linked or embedded objects when loading or saving documents.

## Setting up the Edit menu and the tool bar

An OLE container places OLE commands on its Edit menu. Table 19.3 describes the standard OLE commands. It's not necessary to use all of them, but every container should support at least Insert Object, to let the user add new objects to the current document, and Edit Object, to let the user activate the currently selected object. The *TOleView* class has default implementations for all the commands. It invokes standard dialog boxes where necessary and processes the user's response. All you have to do is add the commands to the Edit menu for each view you derive from *TOleView*.

**Table 19.3** Commands an OLE container places on its Edit menu

| Menu command | Predefined identifier | Command description |
|---|---|---|
| Paste Special | CM_EDITPASTESPECIAL | Lets the user choose from available formats for pasting an object from the Clipboard. |
| Paste Link | CM_EDITPASTELINK | Creates a link in the current document to the object on the Clipboard. |
| Insert Object | CM_EDITINSERTOBJECT | Lets the user create a new object by choosing from a list of available types. |
| Edit Links | CM_EDITLINKS | Lets the user manually update the list of linked items in the current document. |
| Convert | CM_EDITCONVERT | Lets the user convert objects from one type to another. |
| Object | CM_EDITOBJECT | Reserves a space on the menu for the server's verbs (actions the server can take with the container's object). |

If your OLE container has a tool bar, assign it the predefined identifier IDW_TOOLBAR. ObjectComponents must be able to find the container's tool bar if a server asks to display its own tool bar in the container's window. If ObjectComponents can identify the old tool bar, it temporarily replaces it with a new one taken from the server. For ObjectComponents to identify the container's tool bar, the container must use the IDW_TOOLBAR as its window ID, as shown here.

```
TControlBar *cb = new TControlBar(parent);
cb->Attr.Id = IDW_TOOLBAR;          // use this identifier
```

The *TOleFrame::EvAppBorderSpaceSet* method uses the IDW_TOOLBAR for its default implementation. A container can provide its own implementation to handle more complex situations, such as merging with multiple tool bars.

## Loading and saving compound documents

When the user pastes or drops an OLE object into a container, the object becomes data in the container's document. The container must store and load the object along with the rest of the document whenever the user chooses Save or Open from the File menu. The new *Commit* and *Open* methods of *TOleDocument* perform this chore for you. All you have to do is add calls to the base class in your own implementation of *Open* and *Commit*. The code that reads and writes your document's native data remains unchanged.

Because *TOleDocument* is derived from *TStorageDocument* rather than *TFileDocument*, it always creates compound files. *Compound files* are a feature of OLE 2 used to organize the contents of a disk file into separate compartments . You can ask to read or write from any compartment in the file without worrying about where on the disk the

compartment begins or ends. OLE calls the compartments *storages*. The storages in a file can be ordered hierarchically, just like directories and subdirectories. Any storage compartment can contain other sub-storages.

Compound files are good for storing compound documents. When you call *Open* or *Commit*, ObjectComponents automatically creates storages in your file to hold whatever objects the document contains. All the document's native data is saved in the file's root storage. Your existing file data structure remains intact, isolated in a separate compartment. The following code shows how load compound documents.

```
// document class declaration derived from TOleDocument
class _DOCVIEWCLASS TMyDocument : public TOleDocument {
  // declarations
}

// document class implementation
bool
TDrawDocument::Open(int mode, const char far* path) {
  TOleDocument::Open(mode, path);    // load any embedded objects
  ⋮                                  // code to load other document data
}
```

The *TOleDocument::Open* command does not actually copy the data for all the objects into memory. ObjectComponents is smart enough to load the data for particular objects only when the user activates them.

The next code shows how to save compound documents.

```
bool
TMyDocument::Commit(bool force) {
  TOleDocument::Commit(force);      // save the embedded objects
  ⋮                                 // code to save other document data
  TOleDocument::CommitTransactedStorage();    // commit if in transacted mode
}
```

By default, *TOleDocument* opens compound files in transacted mode. *Transacted mode* saves changes in a temporary buffer and merges them with the file only after an explicit command. A revert command discards any uncommitted changes. *Commit* buffers a new transaction. *CommitTransactedStorage* merges all pending transactions.

The opposite of transacted mode is direct mode. *Direct mode* eliminates buffers and makes each change take effect immediately. To alter the default mode, override *TOleDocument::PreOpen*. Omit the *ofTransacted* flag to specify direct mode.

**Note** In order for compound file I/O to work correctly, you need to include the *dtAutoOpen* flag when you register *docflags* in the document registration table.

## 4. Building the container

To build the container, include the right headers, compile with a supported memory model, and link to the ObjectComponents and OLE libraries.

## Including OLE headers

An ObjectComponents program needs the classes, structures, macros, and symbols defined in the header files for the ObjectWindows OLE classes. The following list shows the headers needed for an OLE container that uses the Doc/View model and an MDI frame window.

```
#include <owl/oledoc.h>      // replaces docview.h
#include <owl/oleview.h>     // replaces docview.h
#include <owl/olemdifr.h>    // replaces mdi.h
```

An SDI application includes oleframe.h instead of olemdifr.h.

## Compiling and linking

Containers that use ObjectComponents and ObjectWindows require the large memory model. Link them with the OLE and ObjectComponents libraries.

The integrated development environment (IDE) chooses the right build options when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OWLOCFMK.GEN file found in the EXAMPLES subdirectory. Here, for example, is the makefile that builds the AutoCalc sample program:

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj autocalc.obj
HLP = MYPROGRAM
!include $(BCEXAMPLEDIR)\owlocfmk.gen
```

EXERES and OBJEXE hold the name of the file to build and the names of the object files to build it from. HLP is an optional online Help file. Finally, your makefile should include the OWLOCFMK.GEN file.

Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=l
```

Make, using instructions in OWLOCFMK.GEN, builds a new makefile tailored to your project. The new makefile is called WIN16L*xx*.MAK. The final two digits of the name tell whether the makefile builds diagnostic or debugging versions of the libraries. *01* indicates a debugging version, *10* a diagnostic version, and *11* means both kinds of information are included. The same command then runs the new makefile and builds the program. If you change the command to define MODEL as *d*, the new makefile is WIN16D*xx*.MAK and it builds the program as a DLL.

For more information about how to use OWLOCFMK.GEN, read the instructions at the beginning of MAKEFILE.GEN, found in the EXAMPLES directory.

Table 19.4 shows the libraries an ObjectComponents program links with.

**Table 19.4** Libraries for building ObjectComponents programs

| Large model libraries | DLL import libraries | Description |
|---|---|---|
| OCFWL.LIB | OCFWI.LIB | ObjectComponents |
| OWLWL.LIB | OWLWI.LIB | ObjectWindows |
| BIDSL.LIB | BIDSI.LIB | Class libraries |

**Table 19.4** Libraries for building ObjectComponents programs (continued)

| Large model libraries | DLL import libraries | Description |
| --- | --- | --- |
| OLE2W16.LIB | OLE2W16.LIB | OLE system DLLs |
| IMPORT.LIB | IMPORT.LIB | Windows system DLLs |
| MATHWL.LIB | | Math support |
| CWL.LIB | CRTLDLL.LIB | C run-time libraries |

The ObjectComponents library must be linked first, before the ObjectWindows library. Also, ObjectComponents requires RTTI and exception handling. Do not use compiler command line options that disable these features.

# Turning an ObjectWindows application into an OLE container

Turning an ObjectWindows application into an OLE container requires a few modifications. This list describes them briefly. The sections that follow give more detail for each one.

1. Set up the application.
   - Define an application dictionary object.
   - Modify your application object and implement a new method for it.
2. Register the application.
   - Use registration macros to describe your container.
   - Create a *TOcRegistrar* object to register and run the application.
3. Set up the client window.
   - Use a client window if you have an SDI application and use client windows in your MDI child windows if you have an MDI application.
   - Derive your frame window and client window from the new ObjectWindows OLE classes.
   - Create your frame and client in two steps if you have an SDI application.
   - Create a pair of ObjectComponents objects for each document the application opens.
4. Program the user interface.
   - If you override handlers for certain windows messages, be sure to call the handler in the base class.
   - Set up your menu resource to support menu sharing.
   - Place standard OLE commands on the Edit menu.
   - If you have a tool bar, assign it the standard predefined identifier.
5. Build the application.
   - Include new ObjectWindows OLE headers.
   - Compile and link the application.

By following these steps, you give your ObjectWindows application the following features:

- Linking
- Embedding
- OLE clipboard operations
- Drag and drop operations
- In-place editing
- Tool bar and menu merging
- Compound document storage
- OLE 2 user interface

The following sections expand on each step required to convert your ObjectWindows application into an OLE 2 container. The code excerpts are from the OWLOCF0.CPP sample in the EXAMPLES/OWL/TUTORIAL/OLE directory. The OWLOCF0.CPP sample is based on the STEP10.CPP sample used in the *ObjectWindows Tutorial*. It does not support OLE. OWLOCF1.CPP modifies the first program to create an OLE container.

## 1. Setting up the application

This section describes the changes needed to set up the application for ObjectComponents. The application needs an application dictionary, and the object you derive from *TApplication* must also derive from *TOcModule*.

### Defining an application dictionary object

When a DLL is used by more than one application or process, it must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. For example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* object which corresponds to the current active process. If you turn your application into a DLL server, the application must also maintain a dictionary of the *TApplication* objects created as each new client attaches to the DLL. The DEFINE_APP_DICTIONARY macro provides a simple and unified method for creating an application dictionary object. Insert the following statement with your other static variable declarations.

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

The DEFINE_APP_DICTIONARY macro correctly defines the *AppDictionary* variable regardless of how the application is built. In applications using the static version of ObjectWindows, it simply creates a reference to the existing ObjectWindows application dictionary. For DLL-servers using the DLL version of ObjectWindows, however, the macro declares a instance of the *TAppDictionary* class. It is important to use the name *AppDictionary* when creating your application dictionary object. This allows you to take advantage of the factory template classes for implementing a factory callback function (see "Creating a registrar object").

### Modifying your application class

ObjectWindows provides the mix-in class *TOcModule* for applications that support linking and embedding. Change your application object so it derives from both *TApplication* and *TOcModule* as shown in the following example:

```
// Non-OLE application
class TScribbleApp : public TApplication { /* declarations */ };
```

```
// New declaration of same class
class TScribbleApp : public TApplication, public TOcModule { /* declarations */ };
```

The *TOcModule* object coordinates basic housekeeping chores related to registration and memory management. It also connects your application object to OLE.

Your *TApplication*-derived class must provide a *CreateOleObject* method with the following signature:

```
TUnknown*  CreateOleObject(uint32 options, TDocTemplate* tpl);
```

The method is used by the factory template class. Because containers don't create OLE objects, a container can implement *CreateOleObject* by simply returning 0. As the next chapter explains, servers have more work to do to implement *CreateOleObject*.

```
//
// non-OLE application class
//
class TScribbleApp : public TApplication {
  public:
    TScribbleApp() : TApplication("Scribble Pad") {}

  protected:
    InitMainWindow();
    ⋮
};


//
// New declaration of same class
//
class TScribbleApp : public TApplication, public TOcModule {
  public:
    TScribbleApp() : TApplication(::AppReg["description"]){}
    TUnknown*  CreateOleObject(uint32, TDocTemplate*){ return 0; }

  protected:
    InitMainWindow();
```

## 2. Registering a container

To register an application, you build registration tables with macros. Then you pass the tables to a registrar object to process the information they contain.

### Creating registration tables

OLE requires programs to identify themselves by registering unique identifiers and names. ObjectWindows offers macros that let you build a structure to hold registration information. The structure can then be used when creating the application's instance of *TOcRegistrar*. Here are the commands to create a simple container registration structure:

```
REGISTRATION_FORMAT_BUFFER(100)     // create buffer for expanding macros

BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid, "{9B0BBE60-B6BD-101B-B3FF-86C8A0834EDE}")
  REGDATA(description, "Scribble Pad Container")
END_REGISTRATION
```

The first macro, REGISTRATION_FORMAT_BUFFER, sets the size of a buffer needed temporarily as the macros that are expanded. The REGDATA macro places items in the registration structure, *AppReg*. Each item in *AppReg* is a smaller structure that contains a key, such as *clsid* or *progid*, and a value assigned to the key. The values you assign are case-sensitive strings. The order of keys within the registration table does not matter.

Insert the registration macros after your declaration of the application dictionary. Since the value of the *clsid* key must be a unique number identifying your application, it is recommended that you generated a new value using the GUIDGEN.EXE utility. (The *ObjectWindows Reference Guide* entry for *clsid* explains other ways to generate an identifer.) Of course, modify the value of the *description* key to describe your container.

The *AppReg* structure built in the sample code is an *application registration structure*. A container may also build one or more *document registration structures*. Both structures are built alike, but each contains a different set of keys and values. The keys in an application registration structure describe attributes of the application. A document registration structure describes the type of document an application can create. A document's attributes include the data formats that it can exchange with the clipboard, its file extensions, and its document type name. The OWLOCF1 sample application does not create any document registration structures.

For a list of all the registration keys that a container can use, refer to Table 19.2.

## Creating a registrar object

Every ObjectComponents application needs to create a registrar object to manage all of its registration tasks. Insert the following line after the **#include** statements in your main .CPP file.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TOcRegistrar* instance is created in your *OwlMain* function. Declaring the pointer of type *TPointer<TOcRegistrar>* instead of *TOcRegistrar** ensures that the *TOcRegistrar* instance isdeleted.

**Note** Name the variable *Registrar* to take advantage of the *TOleFactory* template for implementing a factory callback.

The next step is to modify your *OwlMain* function to allocate a new *TOcRegistrar* object to initialize the global pointer *Registrar*. The *TOcRegistrar* constructor requires three parameters: the application's registration structure, the component's factory callback and the command line string that invoked that application.

- The registration structure you create with the registration macros (see the preceding section "Creating registration tables").
- The factory callback you create with an ObjectWindows factory template.

You can write your own callback function from scratch if you prefer, but the templates are much easier to use. For a linking and embedding ObjectWindows application that doesn't use Doc/View, the template class is called *TOleFactory*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.

- The command line string comes from the *GetCmdLine* method of *TApplication*.

Here is the code to create the registrar.

```
int OwlMain(int, char*[])
{
  ⋮
  // create the registrar object
  ::Registrar = new TOcRegistrar(::AppReg, TOleFactory<TScribbleApp>(),
                                 TApplication::GetCmdLine());
  ⋮
}
```

Factories are explained in more detail in the *ObjectWindows Reference Guide*.

After initializing the *Registrar* pointer, your OLE container application must invoke *TOcRegistrar::Run* instead of *TApplication::Run*. For OLE containers, the registrar's *Run* simply invokes the application object's *Run* to create the application's windows and process messages. In a server, however, *TOcRegistrar::Run* does more. Using the registrar's *Run* method in a container makes it easier to modify the application later if you decide to turn it into a server.

Here is the *OwlMain* from OWLOCF1, omitting for clarity the usual **try** and **catch** statements. The lines in bold are the new code.

Before:

```
// Non-OLE OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TScribbleApp().Run();
}
```

After adding the registrar object:

```
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg, TOleFactory<TScribbleApp>(),
                                TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

## 3. Setting up the client window

An ObjectWindows SDI application can use a frame window that does not contain a client window. Similarly, an ObjectWindows MDI application can use MDI child windows that do not contain a client window. Omitting the client window makes it

harder to convert the application from one kind of frame to another—SDI, MDI, or decorated frame. It is also awkward when building OLE 2 applications. For example, it is easier for a container's main window to make room for a server's tool bar if the container owns a client window. To take full advantage of the ObjectWindows OLE classes, your application must use a client window. For more information about using client windows, see the *ObjectWindows Tutorial*.

## Inheriting from OLE classes

ObjectWindows provide several classes that include default implementations for many OLE operations. To adapt an existing ObjectWindows program to OLE, change its derived classes to inherit from the OLE classes. For a list of the OLE classes and the corresponding classes they replace, see Table 19.1.

The *TOleFrame* and *TOleMDIFrame* classes both derive from decorated window classes. The OLE 2 user interface requires that containers be prepared to handle tool bars and status bars. Even if a container has no such decorations, servers might need to display their own in the container's window. The OLE window classes handle those negotiations for you. The following code shows how to change the declaration for a client window. Boldface type highlights the changes.

Before:

```
// Pre-OLE declaration of a client window
class TScribbleWindow : public TWindow {
  ⋮    // declarations
};
DEFINE_RESPONSE_TABLE1(TScribbleWindow, TWindow);
```

After changing the declaration to derive from an OLE-enabled class:

```
// New declaration of the same window class
class TScribbleWindow : public TOleWindow {
  ⋮    // declarations
};
DEFINE_RESPONSE_TABLE1(TScribbleWindow, TOleWindow);
```

## Delaying the creation of the client window in SDI applications

ObjectWindows applications create their main window in the *InitMainWindow* method of the *TApplication*-derived class. Typically, SDI applications also create their initial client window in the *InitMainWindow* function. The following code shows the typical sequence.

```
void
TDrawApp::InitMainWindow()
{
  // Construct the decorated frame window
  TDecoratedFrame* frame = new TDecoratedFrame(0, "Drawing Pad",
                                            new TDrawWindow(0), true);
  ⋮    // more declarations to init and set the main window
}
```

When used in the OLE frame and client classes, however, that sequence presents a timing problem for OLE. The OLE client window must be created after the OLE frame

has initialized its variables pointing to ObjectComponents classes. To meet this requirement, an SDI OLE application should create only the frame window in the *InitMainWindow* function. Create the client window in the *InitInstance* method of your application class. Boldface type highlights the changes.

```
void
TDrawApp::InitMainWindow()
{
  // construct the decorated frame window
  TOleFrame* frame = new TOleFrame("Drawing Pad", 0, true);

  :    // more declarations to init and set the main window
}


void
TDrawApp::InitInstance()
{
  TApplication::InitInstance();

  // create and set client window
  GetMainWindow()->SetClientWindow(new TDrawWindow(0));
}
```

## Creating ObjectComponents view and document objects

For every client window capable of having linked or embedded objects, you must create a *TOcDocument* object to manage the embedded OLE objects, and a *TOcView* object to manage the presentation of the OLE objects. The *CreateOcView* method from the *TOleWindow* class creates both the container document and the container view. Add a call to *CreateOcView* in the constructor of your *TOleWindow*-derived class.

```
// Pre-OLE declaration of a client window constructor
TScribbleWindow::TScribbleWindow(TWindow* parent, char far* filename)
: TWindow(parent, 0, 0)
{
  :
}


// New declaration of client window constructor
TScribbleWindow::TScribbleWindow(TWindow* parent, char far* filename)
: TOleWindow(parent, 0)
{
  :

  // Create TOcDocument object to hold OLE parts
  // and TOcView object to provide OLE services.
  CreateOcView(0, false, 0);
}
```

Notice that unlike the *TWindow* constructor, the *TOleWindow* constructor does not require a *title* parameter. It is unnecessary because *TOleWindow* is always the client of a

frame. *TWindow*, on the other hand, can be used as a non-client window—a pop-up, for example.

# 4. Programming the user interface

The next set of adaptations provide standard OLE user interface features such as menu merging and drag and drop.

## Handling OLE-related messages and events

ObjectComponents notifies your application's windows of OLE-related events by sending the WM_OCEVENT message. The ObjectWindows OLE classes provide default handlers for the various WM_OCEVENT event notifications. Furthermore, the ObjectWindows classes also process a few standard Windows messages to add additional features of the standard OLE user interface. For example, if a user double-clicks within the client area of your container window, a handler in *TOleWindow* checks whether the click occurred over an embedded object and, if so, activates the object. Similarly, the *TOleWindow::EvPaint* method causes each embedded object to draw itself. Table 19.5 lists the methods implemented by the client window (*TOleWindow*) and frame window(*TOleFrame, TOleMDIFrame*) classes. If you override these handlers in your derived class you must invoke the base class version.

**Table 19.5** Standard message handlers providing OLE functionality

| Method | Message | Class | Description |
|---|---|---|---|
| EvSize | WM_SIZE | Frame | Notifies embedded servers of the size change. |
| EvTimer | WM_TIMER | Frame | Invokes *IdleAction* so that DLL servers can carry out command enabling. |
| EvActivateApp | WM_ACTIVATEAPP | Frame | Notifies embedded servers about being activated. |
| EvLButtonDown | WM_LBUTTONDOWN | Client | Deactivates any in-place active object. |
| EvRButtonDown | WM_RBUTTONDOWN | Client | Displays pop-up verb menu if cursor is on an embedded object. |
| EvLButtonDblClk | WM_LBUTTONDBLCLK | Client | Activates any embedded object under the cursor. |
| EvMouseMove | WM_MOUSEMOVE | Client | Allows user to move or resize an embedded object. |
| EvLButtonUp | WM_LBUTTONUP | Client | Informs the selected object of position or size changes. |
| EvSize | WM_SIZE | Client | Informs *TOcView* object that window has changed size. |
| EvMdiActivate | WM_MDIACTIVATE | Client | Informs *TOcView* object that window has changed size. |
| EvMouseActivate | WM_MOUSEACTIVATE | Client | Forwards the message to the top-level parent window and returns the code to activate the client window. |
| EvSetFocus | WM_SETFOCUS | Client | Notifies any in-place server of focus change. |
| EvSetCursor | WM_SETCURSOR | Client | Changes cursor shape if within an embedded object. |

**Table 19.5** Standard message handlers providing OLE functionality (continued)

| Method | Message | Class | Description |
|---|---|---|---|
| EvDropFiles | WM_DROPFILES | Client | Embeds dropped file(s). |
| EvPaint | WM_PAINT | Client | Causes embedded objects to paint. |
| EvCommand | WM_COMMAND | Client | Processes command IDs of verbs. |
| EvCommandEnable | WM_COMMANDENABLE | Client | Processes command IDs of verbs. |

In some cases, you might need to know what action the base class handler took before you decide what to do in your overriding handler. This is particularly true for mouse-related messages. If the base class handled a double-click action, for example, the user intended the action to activate an object and you probably don't want your code to reinterpret the double-click as a different command. The code that follows shows how to coordinate with a base class handler. These three procedures let the user draw on the surface of the client window with the mouse.

```
void
TMyClient::EvLButtonDown(uint modKeys, TPoint& pt)
{
  if (!Drawing) {
    SetCapture()
    Drawing = true;
    :     // additional GDI calls to display drawing
  }
}

void
TMyClient::EvMouseMove(uint modKeys, TPoint& pt)
{
  if (Drawing) {
    :     // additional GDI calls to display drawing
  }
}

void
TMyClient::EvLButtonUp(uint modKeys, TPoint& pt)
{
  if (Drawing) {
    Drawing = false;
    ReleaseCapture();
  }
}
```

As an OLE container, however, the client window may contain embedded objects. Mouse events performed on these objects should not result in any drawing operation. This code shows the handlers updated to allow and check for OLE related processing. Boldface type highlights the changes.

```
void
TMyClient::EvLButtonDown(uint modKeys, TPoint& pt)
{
  TOleWindow::EvLButtonDown(modKeys, pt);
```

```
    if (!Drawing && !SelectEmbedded()) {
      SetCapture()
      Drawing = true;
      :    // additional GDI calls to display drawing
    }
}

void
TMyClient::EvMouseMove(uint modKeys, TPoint& pt)
{
    TOleWindow::EvMouseMove(modKeys, pt);

    if (Drawing && !SelectEmbedded()) {
      :    // additional GDI calls to display drawing
    }
}

void
TMyClient::EvLButtonUp(uint modKeys, TPoint& pt)
{
    if (Drawing && !SelectEmbedded()) {
      Drawing = false;
      ReleaseCapture();
    }

    TOleWindow::EvLButtonUp(modKeys, pt);
}
```

The *SelectEmbedded* method is inherited from *TOleWindow*. It returns **true** if an embedded object is currently being moved. The client window calls it to determine whether a mouse message has already been processed by the OLE base class.

Typically, your derived class must call the base class handlers before processing any event or message. The *EvLButtonUp* handler, however, calls the base class last. Doing so allows the handler to rely on *SelectEmbedded* which is likely to be reset after *TOleWindow* processes the mouse-up message.

## Supporting menu merging

The menu bar of an OLE container with an active object is composed of individual pieces from the normal menus of both the container and server. The container contributes pop-up menus dealing with the application frame or with documents. The server, on the other hand, provides the Edit menu, the Help menu, and any menus that let the user manipulate the activated object.

OLE divides the top-level menus of a menu bar into six groups. Each group is a set of contiguous top-level drop-down menus. Each group is made up of zero or more pop-up menus. The menu groups are named File, Edit, Container, Object, Window, and Help. The group names are for convenience only. They suggest a common organization of related commands, but you can group the commands any way you like.

When operating on its own, a container or server provides the menus for all of the six groups. During an in-place edit session, however, the container retains control of the File, Container and Window groups while the server is responsible for the Edit, Object, and Help groups.

The *TMenuDescr* class automatically handles all menu negotiations between the server and the container. You simply identify the various menu groups within your menu resource, and ObjectWindows displays the right ones at the right times.

To indicate where groups begin and end in your menu resource, insert SEPARATOR menu items between them. Remember to mark all six groups even if some of them are empty. The *TMenuDescr* class scans for the separators when loading a menu from a resource. It removes the separators found between top-level entries and builds a structure which stores the number of pop-up menus assigned to each menu group. This information allows ObjectWindows to merge the server's menu into your container's menu bar.

The following menu resource script, taken from STEP10.RC in the *ObjectWindows Tutorial* tutorial, illustrates defining a simple application menu before it is divided into groups.

```
COMMANDS MENU
{
  pop-up "&File"
  {
    MENUITEM "&New",     CM_FILENEW
    MENUITEM "&Open",    CM_FILEOPEN
    MENUITEM "&Save",    CM_FILESAVE
    MENUITEM "Save &As", CM_FILESAVEAS
  }

  pop-up "&Tools"
  {
    MENUITEM "Pen &Size",  CM_PENSIZE
    MENUITEM "Pen &Color", CM_PENCOLOR
  }

  pop-up "&Help"
  {
    MENUITEM "&About",     CM_ABOUT
  }
}
```

The File menu entry belongs to the OLE File menu group. The Tools menu allows the user to edit the application's document, so it belongs to the Edit group. This application does not contain any menus belonging to the Object, Container, or Window group. And finally, the Help menu belongs to the Help group.

The following code is a modified version of the same menu resource with SEPARATOR dividers inserted to indicate where one group stops and the next begins. Boldface type highlights the changes.

```
COMMANDS MENU
{
```

```
  pop-up "&File"
  {
    MENUITEM "&New",     CM_FILENEW
    MENUITEM "&Open",    CM_FILEOPEN
    MENUITEM "&Save",    CM_FILESAVE
    MENUITEM "Save &As", CM_FILESAVEAS
  }

  MENUITEM SEPARATOR         // end of File group, beginning of Edit group

  pop-up "&Tools"
  {
    MENUITEM "Pen &Size",  CM_PENSIZE
    MENUITEM "Pen &Color", CM_PENCOLOR
  }

  MENUITEM SEPARATOR         // end of Edit group, beginning of Container group
  MENUITEM SEPARATOR         // end of Container group, beginning of Object group
  MENUITEM SEPARATOR         // end of Object group, beginning of Window group
  MENUITEM SEPARATOR         // end of Window group, beginning of Help group

  pop-up "&Help"
  {
    MENUITEM "&About",     CM_ABOUT
  }
}
```

Insert separators in your application's menu to indicate the various menu groups. Then modify your code to use the *SetMenuDescr* method when assigning your frame window's menu. This example shows the menu assignment before and after adding menu merging. Boldface type highlights the changes.

Before:

```
// original menu assignment
void
TScribbleApp::InitMainWindow()
{
  TDecoratedFrame* frame;
  :    // Initialize frame and decorations etc. etc.

  // Assign frame's menu
  frame->AssignMenu("COMMANDS");
}
```

After including group indicators in the menu:

```
void
TScribbleApp::InitMainWindow()
{
  TOleFrame* frame;
  :    // Initialize frame and decorations etc. etc.

  // Assign frame's menu
```

```
  frame->SetMenuDescr(TMenuDescr("COMMANDS"));
}
```

Instead of using separators to show which drop-down menus belong to each group, you can use the *TMenuDescr* constructor whose parameters accept a count for each group. For more details, see the description of the *TMenuDescr* constructors in the *ObjectWindows Reference Guide*.

## Updating the Edit menu

An OLE container places OLE commands on its Edit menu. Table 19.3 on page 312 lists all of the commands. The *TOleWindow* class has default implementations for all of them. It invokes standard dialogs boxes where necessary and processes the user's response. All you have to do is add the commands to the Edit menu of your frame window. It's not necessary to support all six commands, but every container should support at least CM_EDITINSERTOBJECT, to let the user add new objects to the current document, and CM_EDITOBJECT, to let the user choose verbs for the currently selected object.

ObjectWindows defines standard identifiers for the OLE Edit menu commands in owl/oleview.rh. Update your resource file to include the header file and use the standard identifiers to put OLE commands on the Edit menu.

```
#include <owl/oleview.rh>
#include <owl/edit.rh>

COMMANDS MENU
{
  :   // File menu goes here

  MENUITEM SEPARATOR
  pop-up "&Edit"
  {
    MENUITEM "&Undo\aCtrl+Z",          CM_EDITUNDO
    MENUITEM Separator
    MENUITEM "&Cut\aCtrl+X",           CM_EDITCUT
    MENUITEM "C&opy\aCtrl+C",          CM_EDITCOPY
    MENUITEM "&Paste\aCtrl+V",         CM_EDITPASTE
    MENUITEM "Paste &Special...",      CM_EDITPASTESPECIAL
    MENUITEM "Paste &Link",            CM_EDITPASTELINK
    MENUITEM "&Delete\aDel",           CM_EDITDELETE
    MENUITEM SEPARATOR
    MENUITEM  "&Insert Object...",     CM_EDITINSERTOBJECT
    MENUITEM  "&Links...",             CM_EDITLINKS
    MENUITEM  "&Object",               CM_EDITOBJECT
    MENUITEM SEPARATOR
    MENUITEM  "&Show Objects",         CM_EDITSHOWOBJECTS
  }
  :   // other menus go here
}
```

### Assigning a tool bar ID

If your OLE container has a tool bar, assign it the predefined identifier IDW_TOOLBAR. ObjectWindows must be able to find the container's tool bar if a server needs to display its own tool bar in the container's window. If ObjectWindows can identify the old tool bar, it temporarily replaces it with the new one taken from the server. For ObjectWindows to identify the container's tool bar, the container must use the IDW_TOOLBAR as its window ID.

```
TControlBar* cb = new TControlBar(parent);
cb->Attr.Id = IDW_TOOLBAR;
```

The *TOleFrame::EvAppBorderSpaceSet* method uses the IDW_TOOLBAR for its default implementation. A container can provide its own implementation to handle more complex situations, such as merging with multiple tool bars.

## 5. Building a container

A container must include OLE ObjectWindows headers, compile with a supported memory model, and link to the right libraries.

### Including OLE headers

ObjectWindows provides OLE-related classes, structures, macros and symbols in various header files. The following list shows the headers needed for an OLE container using an SDI frame window.

```
#include <owl/oleframe.h>
#include <owl/olewindo.h>
#include <ocf/ocstorag.h>
```

An MDI application includes olemdifr.h instead of oleframe.h.

### Compiling and linking

ObjectWindows containers and servers must be compiled with the large memory model. They must be linked with the OLE, ObjectComponents, and ObjectWindows libraries. Follow the same steps described on page 314 for building a Doc/View application.

# Turning a C++ application into an OLE container

If you are writing a new program, consider using ObjectWindows to save yourself some work. The ObjectWindows Library contains built-in code that automatically performs some tasks common to all ObjectComponents programs. Programs that don't use ObjectWindows must undertake these chores for themselves.

If you are writing a new program, consider using the AppExpert and ObjectWindows to save yourself some work. But ObjectComponents works well in straight C++ programs without ObjectWindows, as well.

This list briefly describes the changes needed for turning a C++ application into an ObjectComponents container. The sections that follow explain each step in more detail.

1. Register the application.
   - Build an application registration table with registration macros.
   - Create a registrar object and call its *CreateOcApp* function.
   - Create a *TOleAllocator* object to initialize the OLE libraries.
2. Create a view window to display an open document.
   - Create, resize, and destroy the view window together with the main window.
   - Create a pair of helper objects, *TOcDocument* and *TOcView*, to manage the OLE side of a compound document.
   - Make the view window handle the WM_OCEVENT message.
   - Write handlers for selected ObjectComponents view events.
   - Draw the compound document in the view window.
   - Activate and deactivate objects in response to user actions.
3. Program the main window to handle OLE commands and events.
   - Pass the *TOcApp* object a handle to the main window.
   - Make the main window handle the WM_OCEVENT message.
   - Write handlers for selected ObjectComponents application events.
   - Add OLE commands (such as Insert Object) to the Edit menu and write handlers for them.
4. Build the program.
   - Include ObjectComponents headers.
   - Compile with the large memory model. Link to the OLE and ObjectComponents libraries.

The sections that follow illustrate each step using examples from the programs in the EXAMPLES/OCF/CPPOCF directory. The source files titled CPPOCF0 contain a windows application that does not support OLE. CPPOCF1 modifies the first program to make it an OLE container. The code samples for this discussion come from CPPOCF1. The same directory also contains CPPOCF2, an OLE server. Chapter 20 uses CPPOCF2 to illustrate making an OLE server without ObjectWindows.

CPPOCF1 is a simple application that supports basic container functions: registering the application, creating objects to initialize a new document, and embedding an object in the document. For ideas about implementing other features, you might want to look at the source code for ObjectWindows OLE classes such as *TOleWindow* and *TOleView*.

The explanations that follow do not describe all the differences in the source code from CPPOCF0 to CPPOCF1. The omit details that are not specific to ObjectComponents and OLE, such as calling *RegisterClass* for a new child window.

# 1. Registering a container

Giving the system the information it needs about your container takes three steps: building a registration table, passing the table to a registrar object, and creating a memory allocator object.

## Building a registration table

A container uses the registration macros to build a registration table describing the application. A container does not need to create document registration tables except to support being a link source. (When a container is a link source, it allows other containers to create links to objects in its own documents.)

Here is the registration table from CPPOCF1:

```
REGISTRATION_FORMAT_BUFFER(100)
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,        "{8646DB80-94E5-101B-B01F-00608CC04F66}")
  REGDATA(progid,       APPSTRING ".Application.1")
  REGDATA(description,  "Sample container")
END_REGISTRATION
```

The application's header file includes this line:

```
#define APPSTRING   "CppOcf1"
```

The *progid* string is therefore "CppOcf1.Application.1."

The registration macros build a structure of type *TRegList*. Each entry in the structure contains a key, such as *clsid* or *progid*, and a value assigned to the key. Internally ObjectComponents finds the values by searching for the keys. The order in which the keys appear does not matter. For more information about the keys a container can choose to register, refer to Table 19.2. For more information about registration in ObjectComponents, see "Understanding registration" on page 372.

## Creating the registrar object

The registrar object records application information in the system registration database, processes any OLE switches on the application's command line, and notifies OLE that the server is running. CPPOCF1 declares a static pointer for the registrar object:

```
TOcRegistrar* OcRegistrar = 0;
TOcApp*       OcApp       = 0;
```

The second variable, *OcApp*, points to the connector object that implements OLE interfaces for the application to communicate with OLE. The registrar creates the *TOcApp* object in *WinMain*.

Create the registrar as you initialize the application in *WinMain*. Instead of entering a message loop, call the registrar's *Run* method. When *Run* returns, the application is ready to shut down. Delete the registrar before you quit. This excerpt from the CPPOCF1 *WinMain* function shows all the steps.

```
int PASCAL
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  char far* lpCmdLine, int nCmdShow)
{
  try {
    TOleAllocator allocator(0); //required for OLE2
    MSG msg;

    // Initialize OCF objects
```

```
    OcRegistrar = new TOcRegistrar(::AppReg, 0,
                                  string(lpCmdLine), 0);
    OcRegistrar->CreateOcApp(OcRegistrar->GetOptions(), OcApp);

    ⋮        // per-instance and per-task initialization code goes here

    // Standard Windows message loop
    while (GetMessage(&msg, 0, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  catch (TXBase& xbase) {
    MessageBox(GetFocus(), xbase.why().c_str(), "Exception caught", MB_OK);
  }

  // free the registrar object
  delete OcRegistrar;
  return 0;
}
```

The *TOcRegistrar* constructor takes four parameters:

- *::AppReg*, is the application registration structure already built with the registration macros. (See the section "Building a registration table.")
- *ComponentFactory* is a callback function described in the next section.

  The callback is responsible for creating any of the application's OLE components, including the application itself, as required. The callback contains the application's message loop, as well.
- *cmdLine* is a *string* object holding the application's command line.

  The registrar searches the command line for OLE-related switches such as -Automation or -Embedding, and it sets internal running mode flags accordingly.
- 0 is a null pointer to a document list.

  Because CPPOCF1 does not register any document types, this list is empty. A container registers document types to support being a link source. For more information about document lists, see "Creating the document list" on page 361.

Besides recording information in the registration database, the registrar object also creates the *TOcApp* connector object when you call *CreateOcApp*.

## Creating a memory allocator

The beginning of the *WinMain* procedure creates a *TOleAllocator*:

```
TOleAllocator allocator(0);      // use default memory allocator
```

The allocator's constructor initializes the OLE libraries and its destructor releases them when the object goes out of scope. Passing 0 to the constructor tells it to let OLE use its standard memory functions whenever allocating memory on behalf of this application.

## 2. Creating a view window

ObjectComponents imposes one design requirement: a compound document must have its own window, separate from the application's main window. To keep the distinction clear, we'll call the main window the *frame* window, because it uses the WS_THICKFRAME style and has a visible border on the screen. The second window has no visible border. We'll call it the *view* window because that is where the application displays its data. The view window always exactly fills the frame window's client area, so from the user's point of view the frame window appears to be the only window. ObjectComponents needs the view window, though, because it expects to send some event messages to the application and some to the view. (View windows are sometimes called *client* windows, too.)

In an SDI application like the CPPOCF1 sample program, the frame window controls the view window. When the frame window receives a WM_SIZE message, it moves the view to keep it aligned with the frame's client area. When it receives WM_CLOSE, it destroys both itself and the view window.

In an MDI application, each child window creates its own view. The child window does what the SDI frame does: creates and manages a view for the document it displays.

### Creating, resizing, and destroying the view window

Before creating the view window, the application must first register a class for the view window. CPPOCF1 registers both classes in *InitApplication*.

CPPOCF1 creates the view window in its factory because the factory is in charge of creating new documents on request. The code for the view window, as you'll see, connects the new document to OLE by creating some ObjectComponents helper objects. The factory calls this function to create the view window:

```
HWND CreateViewWindow(HWND hwndParent)
{
  HWND hwnd = CreateWindow(VIEWCLASSNAME, "",
    WS_CHILD | WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_VISIBLE | WS_BORDER,
    10, 10, 300, 300,
    hwndParent, (HMENU)1, HInstance, 0);
  return hwnd;
}
```

CPPOCF1 resizes and destroys the view window when the frame window receives WM_SIZE and WM_CLOSE messages.

```
void
MainWnd_OnSize(HWND hwnd, UINT /*state*/, int /*cx*/, int /*cy*/)
{
  if (IsWindow(HwndView)) {
    TRect rect;
    GetClientRect(hwnd, &rect);
    MoveWindow(HwndView, rect.left, rect.top, rect.right, rect.bottom, true);
  }
}
```

```
void
MainWnd_OnClose(HWND hwnd)
{
  if (IsWindow(HwndView))
    DestroyWindow(HwndView);
  DestroyWindow(hwnd);
}
```

The view window always fills the frame window's client area exactly. If the user opens and closes documents or embeds objects, the changes show up in the view window.

## Creating a TOcDocument and TOcView

If the user embeds several objects in the container's view window, they all become part of a single compound document. If the container supports file I/O, then the user can save and load different documents.

For every document the container opens or creates, it needs one view window and two helper objects: *TOcDocument* and *TOcView*. The document helper manages the collection of objects inserted in the document. The view helper connects the document to OLE. More specifically, it implements interfaces that OLE can call to communicate with the document. When OLE tells the view object that something noteworthy has occurred, the view object sends a message to the view window. (The next two sections show how to handle the messages.)

The sample CPPOCF1 container declares two global pointers to hold the two helper objects. (A program that opens more than one document at a time needs more than one pair of variables.)

```
TOcDocument*  OcDoc      = 0;
TOcView*      OcView     = 0;
```

CPPOCF1 creates and destroys the two helpers when it creates and destroys the view window that displays the document.

```
bool
ViewWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  OcDoc  = new TOcDocument(*OcApp);   // create document helper
  OcView = new TOcView(*OcDoc);       // create view connector
  if (OcView)
    OcView->SetupWindow(hwnd);        // attach view to window
  return true;
}

void
ViewWnd_OnDestroy(HWND /*hwnd*/)
{
  :  // code to de-activate objects goes here (explained later)

  if (OcView)
    OcView->ReleaseObject();     // do not delete the view; it is a COM object
  OcDoc->Close();                // release the server for each embedded object
```

```
  delete OcDoc;                // delete the document helper; it is not a COM object
}
```

The WM_CREATE message handler for the view window creates both helpers and then calls *OcView->SetupWindow*. The *SetupWindow* method tells the *TOcView* object where to send event messages. In this case, it sends messages to *hwnd*, the view window. The view window now receives WM_OCEVENT messages.

When the view window is destroyed, it makes three calls to dispose of the helper objects. *OcView->ReleaseObject* signals that the view window is through with the *TOcView* connector object. You shouldn't call **delete** for a *TOcView* object because the OLE system might still need more information before it allows the view to shut down. *ReleaseObject* tells the *TOcView* object that you don't need it any longer. The view subsequently destroys itself as soon as all other OLE clients finish with it, as well. The *TOcView* destructor is protected to prevent you from calling it directly.

(A container document only has other OLE clients if it registers support for being a link source. In that case, other applications can create links to objects in the container's document.)

The *TOcDocument* object, on the other hand, is not a connector object and so you can destroy it with **delete** in the usual way. First, however, you should call *Close* to release the server applications that OLE may have invoked to support each linked or embedded object.

## Handling WM_OCEVENT

Because the *TOcView::SetupWindow* method bound the *TOcView* connector to the view window, the connector sends its event notification messages to the window. All ObjectComponents events are sent in the WM_OCEVENT message, so the view window procedure must respond to WM_OCEVENT.

```
long CALLBACK _export
ViewWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    : // other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  ViewWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}
```

The HANDLE_MSG message cracker macro for WM_OCEVENT is defined in the ocf/ocfevx.h header. The same header also defines a another cracker for use in the WM_OCEVENT message handler.

```
// Subdispatch OC_VIEWxxxx messages
long
ViewWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    // insert an event cracker for each OC_VIEWxxxx message you want to handle
    HANDLE_OCF(hwnd, OC_VIEWPARTINVALID, ViewWnd_OnOcViewPartInvalid);
  }
```

```
    return true;
}
```

The WM_OCEVENT message carries an event ID in its *wParam*, just as WM_COMMAND messages carry command IDs. OC_VIEWPARTINVALID is one possible event, indicating that it is time to repaint a linked or embedded object. The HANDLE_OCF macro calls the handler you designate for each ObjectComponents event, just as HANDLE_MSG calls the handler for for a window message.

CPPOCF1 chooses to handle only the OC_VIEWPARTINVALID message. To handle others, add one HANDLE_OCF macro for each event ID.

A list of all the ObjectComponents messages appears in Tables 18.5 and 18.6.

## Handling selected view events

Each HANDLE_OCF macro calls a different handler function. In the example, the handler function is called *ViewWnd_OnOcViewPartInvalid*. ObjectComponents sends this message to a container when one of the OLE data objects in its document needs to be repainted.

```
bool
ViewWnd_OnOcViewPartInvalid(HWND hwnd, TOcChangeInfo far& changeInfo)
{
  HDC dc = GetDC(hwnd);
  SetMapMode(dc, MM_ANISOTROPIC);
  SetWindowOrg(dc, 0, 0);
  SetViewportOrg(dc, 0, 0);
  RECT rect = part.GetRect();
  LPtoDP(dc, (POINT*)&rect, 2);
  InvalidateRect(hwnd, &rect, true);
  ReleaseDC(hwnd, dc);
  return true;
}
```

The *TOcPart* parameter represents the object that needs painting. ObjectComponents creates a *TOcPart* object for every linked or embedded object in a container document. CPPOCF1 handles this message by asking the part for its coordinates and invalidating that part of its client area. The *InvalidateRect* command results in a WM_PAINT message, and the *ViewWnd_OnPaint* procedure responds by drawing the document.

## Painting the document

Painting a compound document requires two steps: drawing the container's own data and drawing all the linked or embedded objects. Here's the basic frame for a paint procedure:

```
void
ViewWnd_OnPaint(HWND hwnd)
{
  PAINTSTRUCT ps;
  HDC dc = BeginPaint(hwnd, &ps);
  //
  // Do your regular painting here
```

```
  //

  // Now draw embedded objects
  ViewWnd_PaintParts(hwnd, dc, false);
  EndPaint(hwnd, &ps);
}
```

The code for *ViewWnd_PaintParts* is the same in most applications.

```
bool
ViewWnd_PaintParts(HWND hwnd, HDC dc, bool metafile)
{
  // get logical coordinates of area to draw
  TRect clientRect;
  GetClientRect(hwnd, &clientRect);
  TRect logicalRect = clientRect;
  DPtoLP(dc, (POINT*)&logicalRect, 2);

  // loop through all the parts and draw each one
  ViewData& viewData = GetViewData(hwnd);
  for (TOcPartCollectionIter i(viewData.OcDoc->GetParts()); i; i++) {
    TOcPart& part = *i.Current();
    if (part.IsVisible(logicalRect)) {
      TRect rect = part.GetRect();
      part.Draw(dc, rect, clientRect, asDefault);
      if (metafile)
        continue;

      // If an object is selected, draw whatever mark indicates that state
      if (part.IsSelected()) {
        // Draw some XOR rectangle around 'rect'
      }
    }
  }
  return true;
}
```

CPPOCF1 is a very simple container. Because it holds only one embedded object at a time, it doesn't really have to create a loop to handle painting multiple parts. If it expanded to permit multiple objects, however, it would not have to change its paint procedure.

The *TOcPart* class manages linked or embedded objects in a container document. The *TOcPart::Draw* method asks the server to draw its object. The *Draw* method does not need to be told the position of the object. The *TOcPart* object receives that information when it is constructed, as you will see in "Handling selected application events."

The **for** loop creates an iterator object to enumerate all the parts in the document. The **++** operator advances the iterator to point to successive parts. The expression **i.Current()* evaluates to a different part each time through the loop.

## Activating and deactivating objects

After embedding an object into a compound document, the user might decide to edit the object. In most containers, the user activates an object by double-clicking it. CPPOCF1 does not support activating objects, but the code to do it is straightforward. You intercept WM_LBUTTONDBLCLK messages, check the mouse coordinates, and if they fall on an object you activate it.

To enumerate the document's embedded and linked objects, use a **for** loop with a *TOcPartCollectionIter* object, as the paint procedure does to draw all the parts. To find the coordinates of an object, call *TOcPart::GetRect*. To activate a part, call *TOcPart::Activate*.

Before a container closes a compound document, it should always check that no object is activated. CPPOCF1 includes this loop in the WM_DESTROY handler of its view window:

```
for (TOcPartCollectionIter i(OcDoc->GetParts()); i; i++) {
  TOcPart& p = *i.Current();
  p.Activate(false);
}
```

Passing **false** to *Activate* terminates any editing session.

# 3. Programming the main window

The view window manages tasks related to a single document. It opens and closes the document and draws it on the screen. The frame window manages tasks for the whole application. It responds to menu commands, and it creates and destroys the view window.

## Creating the main window

When the application creates its main window, it must bind the window to its *TOcApp* object. (The *TOcApp* object was created in the factory callback function.)

```
bool
MainWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  HwndMain = hwnd;
  if (OcApp)
    OcApp->SetupWindow(hwnd);
  return true;
}
```

The *TOcApp* object sends messges about OLE events to the application's main window. *SetupWindow* tells the *TOcApp* where to direct its event notifications.

## Handling WM_OCEVENT

*TOcApp* sends event notifications in the WM_OCEVENT message. Like the view window, the frame window also must handle WM_OCEVENT. The frame window receives notification of events that concern the application as a whole, not just a particular document. The frame window procedure sends WM_OCEVENT messages to

a handler that identifies the event and calls the appropriate handler routine. Both routines closely resemble the corresponding code for the view window.

```
// Standard message-handler routine for main window
long CALLBACK _export
MainWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    ⋮ // other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  MainWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}

// Subdispatch OC_... messages
long
MainWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    HANDLE_OCF(hwnd, OC_VIEWTITLE, MainWnd_OnOcViewTitle);
  }
  return true;
}
```

## Handling selected application events

The only ObjectComponents event that CPPOCF1 handles in its main window is OC_VIEWTITLE. This message is sent when a server engaged in open editing wants to use the container's title in its own window caption. The OLE user interface guidelines require the server to show the source of the object it is editing.

```
const char*
MainWnd_OnOcViewTitle(HWND /*hwnd*/)
{
  return APPSTRING;
}
```

Because CPPOCF1 always has only a single document, it returns the name of the application as the title of its view. Because it always has only one view, it can handle the OC_VIEWTITLE event in the main window procedure. Most containers handle this message in the view window and return the name of the application and the name of the document combined in a single string.

## Handling standard OLE menu commands

An OLE container places OLE commands on its Edit menu. Table 19.3 describes the new commands. It's not necessary to use all of them. CPPOCF1 supports one, Insert Object. This command lets users add new objects to the current document.

```
void
MainWnd_OnCommand(HWND hwnd, int id, HWND /*hwndCtl*/, uint /*codeNotify*/)
{
  switch (id) {
    case CM_INSERTOBJECT: {
```

```
      try {
        TOcInitInfo initInfo(OcView);          // begin initializing info structure
        if (OcApp->Browse(initInfo)) {         // show Insert Object dialog box
          TRect rect(30, 30, 100, 100);        // only top and left are used
          new TOcPart(*OcDoc, initInfo, rect); // add new object to document
        }
      }
      catch (TXBase& xbase) {
        MessageBox(GetFocus(), xbase.why().c_str(), "Exception caught", MB_OK);
      }
      break;
    }
    case CM_EXIT: {
      PostMessage(hwnd, WM_CLOSE, 0, 0);
      break;
    }
  }
}
```

The code for inserting, dropping, or pasting an object into a document always begins with a *TOcInitInfo* structure. *TOcInitInfo* holds information describing the object about to be created: what container will receive it, whether to link or embed it, whether it already exists or will be newly created, and if it exists, where the data resides and in what format.

The constructor for *TOcInitInfo* receives a pointer to the view where you want the new object to appear. The next command, *OcApp->Browse,* invokes the standard Insert Object dialog box offering the user a choice of all the objects any server registered in the system can create. When the user chooses one, the *Browse* command places more information in *initInfo.*

The final step to insert a new OLE object is to create a *TOcPart* connector. *TOcPart* implements all the OLE services that a linked or embedded object is required to provide. It plugs into OLE, gets the data for the new object, adds itself to the list of parts in *OcDoc,* and draws itself on the screen in the position given by *TRect.*

For examples showing how to implement other OLE Edit menu commands, look at the source code for event handlers in OWL/OLEWINDO.CPP.

## Building the program

To build the server, you need to include the right headers, use a supported memory model, and link to the right libraries.

### Including ObjectComponents headers

The following list shows the ObjectComponents headers for a container that does not use ObjectWindows.

```
#include <ocf/ocapp.h>     // TOcRegistrar, TOcModule, TOcApp (application connector)
#include <ocf/ocdoc.h>     // TOcDocument (compound document manager)
#include <ocf/ocview.h>    // TOcView (document view connector)
```

```
#include <ocf/ocpart.h>      // TOcPart (linked/embedded object connector)
#include <ocf/ocfevx.h>      // WM_OCEVENT message crackers
```

## Compiling and linking

ObjectComponents applications that do not use ObjectWindows can use either the medium or large memory model. Link them with the OLE and ObjectComponents libraries.

To build CPPOCF0, CPPOCF1, and CPPOCF2, move to the program's directory and type this at the command prompt:

```
make MODEL=l
```

This command builds all three programs using the large memory model.

The make file that builds this example program refers to the OCFMAKE.GEN file. For more information about using OCFMAKE.GEN in your own make files, see page 314.

Chapter

20

# Creating an OLE server

An OLE server is an application that creates and manages data objects for other programs. This chapter explains how to take existing programs and turn them into linking and embedding servers. It describes the steps required for adapting three kinds of programs:

- An ObjectWindows application that uses the Doc/View model
- An ObjectWindows application that does not use the Doc/View model
- A C++ application that does not use ObjectWindows

In addition, this chapter covers these related topics:

- Registration tables
- DLL servers

The ObjectComponents Framework classes support servers as well as containers, and new ObjectWindows classes make this support easily available. The easiest kind of program to convert is one that uses ObjectWindows and the Doc/View model, but ObjectComponents simplifies the task of writing a server application even without the ObjectWindows framework.

OLE applications can also be automation servers. Chapter 21 shows how to automate an application.

## Turning a Doc/View application into an OLE server

Turning a Doc/View application into an OLE server requires only a few modifications, and many of them are the same as the changes required to create a container. If you have already modified your application according to the steps in Chapter 19, then much of your server work is already done.

The following list describes the changes briefly. The sections that follow describe each step in more detail.

1. Connect your application, window, document, and view objects to OLE.
   - Create a *TAppDictionary* object.
   - Derive your application class from *TOcModule* as well as *TApplication*.
   - Derive frame window, document, and view classes from new OLE-enabled classes.
2. Register the application.
   - Build registration tables to describe the application and the types of documents it produces.
   - Create a registrar object and call its *Run* method.
3. Coordinate with the container to draw, load, and save your objects.
   - Tell OLE when an object's appearance changes. A single function call accomplishes this.
   - Add commands to read and write embedded objects in compound documents.
4. Build the application.
   - Include new ObjectWindows OLE headers at the beginning of your source code.
   - Compile using the large memory model. Link with the OLE and ObjectComponents libraries.

That's all you need to do. After performing these steps, your OLE server supports all the following features:

- Source for linking
- Source for drag and drop
- Registration
- Source for embedding
- In-place editing
- Compound document storage

To modify the default behavior ObjectComponents provides for common OLE options, you can additionally override the default handlers for messages that ObjectComponents sends. For a list of event messages, see Tables 18.5 and 18.6.

# 1. Connecting objects to OLE

Your application, window, document, and view objects need to make use of new OLE-enabled classes. The constructor for the application object expects to receive an application dictionary object, so create that first.

## Creating an application dictionary

An application dictionary tracks information for the currently active process. It is particularly useful for DLLs. When several processes use a DLL concurrently, the DLL must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. For example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* corresponding to the currently active client process. If you convert an executable server to a DLL server, it must also maintain a dictionary of the *TApplication* objects representing each of its container clients.

The DEFINE_APP_DICTIONARY macro provides a simple and unified way to create the dictionary object for any type of application, whether it is a container, a server, a DLL, or an EXE. Insert this statement with your other static variables:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

For any application linked to the static version of the DLL library, the macro simply creates a reference to the application dictionary in ObjectWindows. For DLL servers using the DLL version of ObjectWindows, however, it creates an instance of the *TAppDictionary* class.

**Note** Name your dictionary object *AppDictionary* to take advantage of the factory templates such as *TOleDocViewFactory* (as explained later in "Creating a registrar object").

## Deriving the application object from TOcModule

The application object of an ObjectComponents program needs to derive from *TOcModule* as well as *TApplication*. *TOcModule* coordinates some basic housekeeping chores related to registration and memory management. It also connects your application to OLE. More specifically, *TOcModule* manages the connector object that implements COM interfaces on behalf of an application.

If the declaration of your application object looks like this:

```
class TMyApp : public TApplication {
  public:
    TMyApp() : TApplication(){};
  ⋮
};
```

Then change it to look like this:

```
class TMyApp : public TApplication, public TOcModule {
  public:
    TMyApp(): TApplication(::AppReg["appname"], ::Module, &::AppDictionary){};
  ⋮
};
```

The constructor for the revised *TMyApp* class takes three parameters.

- A string naming the application

  *AppReg* is the application's registration table, shown later in "2. Registering a linking and embedding server." The expression *::AppReg["appname"]* extracts a string that was registered to describe the application.

- A pointer to the application module.

  *Module* is a global variable of type *TModule** defined by ObjectWindows.

- The address of the application dictionary.

  *AppDictionary* is the application dictionary object explained in the previous section.

### Inheriting from OLE classes

A server makes the same changes to its OLE classes that a container makes. ObjectWindows wraps a great deal of power in its new window, document, and view classes. To give an ObjectWindows program OLE capabilities, change its derived classes to inherit from OLE classes.

Here are some examples:

```
// old declarations (without OLE)
class TMyDocument: public TDocument    { /* declarations */ };
class TMyView: public TView            { /* declarations */ };
class TMyFrame: public TFrameWindow    { /* declarations */ );

// new declarations (with OLE)
class TMyDocument: public TOleDocument { /* declarations */ };
class TMyView: public TOleView         { /* declarations */ };
class TMyFrame: public TOleFrame       { /* declarations */ );
```

For a complete list of new OLE classes and their non-OLE counterparts, consult Table 19.1 on page 305.

When you change to OLE classes, be sure that those methods in your classes which refer to their direct base classes now use the OLE class names.

```
void TMyView::Paint(TDC& dc, BOOL erase, TRect& rect)
{
  TOleView::Paint(dc, erase, rect);
  // paint the view here
}
```

It is generally safer to allow the OLE classes to handle Windows events and Doc/View notifications. This is particularly true for the *Paint* method and mouse message handlers in classes derived from *TOleView*. *TOleView::Paint* knows how to paint the objects embedded in your document. (Servers are often containers as well, and a server's object might have other objects embedded in it.) Similarly, the mouse handlers of *TOleView* let the user select, move, resize, and activate an OLE object embedded or linked in your document. See Table 19.5 on page 322 for a list of standard message handlers that provide OLE functionality.

## 2. Registering a linking and embedding server

To register your application with OLE, first create registration tables describing the application and the kinds of documents it creates. The tables create structures that you pass to the registrar object when you create it. Call the registrar's *Run* method to start the application.

### Building registration tables

databaseServers implement OLE objects that any container can use. Different servers implement different types of objects. Every type of object a server creates must have a globally unique identifier (GUID) and a unique string identifier. Every server must record this information, along with other descriptive information, in the registration database of the system where it runs. OLE reads the registry to determine which objects

are available, what their capabilities are, and how to invoke the application that creates objects of each type.

ObjectComponents simplifies the task of registration. You call macros to build a table of keys with associated values. ObjectComponents receives the table and automatically performs all registration tasks.

Servers and containers use the same macros for registration, but servers must provide more information than containers. Here are the commands to build the registration tables for a typical server. This example comes from the STEP15.CPP and STEP15DV.CPP file in the EXAMPLES\OWL\TUTORIAL directory of your compiler installation.

```
REGISTRATION_FORMAT_BUFFER(100)         // allow space for expanding macros

BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,      "{5E4BD320-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(appname,    "DrawPad Server")
END_REGISTRATION

BEGIN_REGISTRATION(DocReg)
  REGDATA(progid,     "DrawPad.Drawing.15")
  REGDATA(description,"DrawPad (Step15--Server) Drawing")
  REGDATA(menuname,   "Drawing")
  REGDATA(extension,  "p15")
  REGDATA(docfilter,  "*.p15")
  REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGDATA(insertable, "")
  REGDATA(verb0,      "&Edit")
  REGDATA(verb1,      "&Open")
  REGFORMAT(0, ocrEmbedSource,  ocrContent,  ocrIStorage,  ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent,  ocrMfPict|ocrStaticMed, ocrGet)
  REGFORMAT(2, ocrBitmap,       ocrContent,  ocrGDI|ocrStaticMed, ocrGet)
  REGFORMAT(3, ocrDib,          ocrContent,  ocrHGlobal|ocrStaticMed, ocrGet)
  REGFORMAT(4, ocrLinkSource,   ocrContent,  ocrIStream, ocrGet)
END_REGISTRATION
```

The macros in the example build two structures. The first structure is named *AppReg* and the second is *DocReg*. ObjectComponents uses lowercase strings such as *progid* and *clsid* to name the standard keys to which you assign values. The values you assign are strings, and they are sensitive to case. The order of keys within the registration table doesn't matter. For more about the macros REGDATA, REGFORMAT, and REGDOCFLAGS, see "Understanding registration macros" on page 309. The full set of registration macros is described in the *ObjectWindows Reference Guide*.

The set of keys you place in a structure depends on what OLE capabilities you intend to support and whether the structure holds application information or document information. The macros in the example show the minimum amount of information a server with one type of document should provide.

A server registers program and class ID strings (*progid* and *clsid*) for itself and for every type of document it creates. The IDs must be absolutely unique so that OLE can

distinguish one application from another. The *description* strings appear on the Insert Object dialog box where the user sees a list of objects available in the system.

Table 20.1 briefly describes all the registration keys that can be used by a server that supports linking and embedding. It shows which are optional and which required as well as which belong in the application registration table and which in the document registration table.database

**Table 20.1** Keys a linking and embedding server registers

| Key | In AppReg? | In DocReg? | Description |
|---|---|---|---|
| appname | Yes | Optional | Short name for the application |
| clsid | Yes | Optional | Globally unique identifier (GUID); generated automatically for the *DocReg* structure |
| description | No | Yes | Descriptive string (up to 40 characters) |
| progid | No | Yes for link or embed source | Identifier for program or object type (unique string) |
| menuname | No | Yes | Name of server object for container menu |
| extension | No | Optional | Document file extension associated with server |
| docfilter | No | Yes if not *dtHidden* | Wildcard file filter for File Open dialog box |
| docflags | No | Yes | Options for running the File Open dialog box |
| debugger | No | Optional | Command line for running debugger |
| debugprogid | No | Optional | Name of debugging version (unique string) |
| debugdesc | No | Yes if using debugprogid | Description of debugging version |
| insertable | No | Yes for embedding | Indication that object can be embedded. If omitted, the document is only a link source |
| verb*n* | No | Yes | An action the server can execute with the object |
| format*n* | No | Yes | A clipboard format the server can produce |
| aspectall | No | Optional | Options for showing object in any aspect |
| aspectcontent | No | Optional | Options for showing object's content aspect |
| aspectdocprint | No | Optional | Options for showing object's docprint aspect |
| aspecticon | No | Optional | Options for showing object's icon aspect |
| aspectthumbnail | No | Optional | Options for showing object's thumbnail aspect |
| cmdline | No | Optional | Arguments to place on server's command line |
| path | No | Optional | Path to server file (defaults to current module path) |
| permid | No | Optional | Name string without version information |
| permname | No | Optional | Descriptive string without version information |
| usage | Optional | Optional | Support for concurrent clients |
| language | Optional | No | Language for registered strings (defaults to system's user language setting) |
| version | Optional | No | Major and minor version numbers (defaults to "1.0") |

The table assumes that the server's documents support linking or embedding. For documents that support neither, the server needs to register only *docflags* and *docfilter*.

If the server is also a container or an automation server, then you should also consult the container table (Table 19.2) or the automation table (Table 21.1). Register all the keys that are required in any of the tables that apply to your application.

For more information about individual registration keys, the values they hold, and the macros used to register them, look for the keys by name in the *ObjectWindows Reference Guide*.

The values assigned to keys can be translated to accommodate system language settings. For more about localization, see the sections "Registering localized entries" on page 373 and "Localizing symbol names" on page 397.

Place your registration structures in the source code files where you construct document templates and implement your *TApplication*-derived class. A server always creates only one application registration table (called *AppReg* in the example). The server might create several document registration tables, however, if it creates several different kinds of documents (for example, text objects and chart objects). Each registration table needs a unique *progid* value.

After creating registration tables, you must pass them to the appropriate object constructors. The *AppReg* structure is passed to the *TOcRegistrar* constructor, as described in the section "Creating a registrar object." Document registration tables are passed to the document template constructor.

```
DEFINE_DOC_TEMPLATE_CLASS(TMyOleDocument, TMyOleView, MyTemplate);
MyTemplate myTpl(::DocReg);
```

databaseSome of the information in the document registration table is used only by the document template. The document filter and document flags have to do with documents, not with OLE. Previous versions of OWL passed the same information to the document template as a series of separate parameters. The old method is still supported for backward compatibility, but new programs, whether they use OLE or not, should use the registration macros to supply document template parameters.

Some of the registration macros expand the values passed to them. The REGISTRATION_FORMAT_BUFFER macro reserves memory needed temporarily for the expansion. To determine how much buffer space you need, allow 10 bytes for each REGFORMAT item plus the size of any string parameters you pass to the macros REGSTATUS, REGVERBOPT, REGICON, or REGFORMAT. (Registration macros are described in the *ObjectWindows Reference Guide*.)

## Creating a registrar object

Every ObjectComponents application needs a registrar object to manage all of its registration tasks. In a linking and embedding application, the registrar is an object of type *TOcRegistrar*. At the top of your source code file, declare a global variable holding a pointer to the registrar.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TPointer* template ensures that the *TOcRegistrar* instance is deleted when the program ends.

**Note** Name this variable *Registrar* to take advantage of the factory callback template used in the registrar's constructor.

The next step is to modify your *OwlMain* function to allocate a new *TOcRegistrar* object and initialize the global pointer *Registrar*. The *TOcRegistrar* constructor expects three parameters: the application's registration structure, the component's factory callback and the command-line string that invoked that application. The registration structure is created with the registration macros.

The factory callback is created with a class template. For a linking and embedding ObjectWindows application that uses Doc/View, the template is called *TOleDocViewFactory*. The third parameter, the command-line string, can be obtained from the *GetCmdLine* method of *TApplication*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.

```
int OwlMain(int, char*[])
{
  // Create Registrar object
  ::Registrar = new TOcRegistrar(::AppReg, TOleDocViewFactory<TMyApp>(),
         TApplication::GetCmdLine());
  ⋮
}
```

After initializing the *Registrar* pointer, your OLE container application must invoke the *Run* method of the registrar instead of *TApplication::Run. TRegistrar::Run* calls the factory callback procedure (the one the second parameter points to) and causes the application to create itself. The application enters its message loop, which is actually in the factory callback. The following code shows a sample *OwlMain* before and after adding a registrar object. Boldface type highlights changes.

Before:

```
// Non-OLE OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TMyApp().Run();
}
```

After:

```
// New declaration of OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg,
        TOleDocViewFactory<TMyApp>(),
        TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

The last parameter of the *TOcRegistrar* constructor is the command line string that invoked the application.

## Processing the command line

When OLE invokes a server, it places an -Embedding switch on the command line to tell the application why it has been invoked. The presence of the switch indicates that the user did not launch the server directly. Usually a server responds by keeping its main window hidden. The user interacts with the server through the container. If the –Embedding switch is not present, the user has invoked the server as a standalone application and the server shows itself in the normal way.

When you construct a *TRegistrar* object, it parses the command line for you and searches for any OLE-related switches. It removes the switches as it processes them, so if you examine your command line after creating *TRegistrar* you will never see them.

If you want to know what switches were found, call *IsOptionSet*. For example, this line tests for the presence of a registration switch on the command line:

```
if (Registrar->IsOptionSet(amAnyRegOption))
      return 0;
```

This is a common test in *OwlMain*. If the a command line switch such as -RegServer was set, the application simply quits. By the time the registrar object is constructed, any registration action requested on the command line have already been performed.

Table 20.2 lists all the OLE-related command-line switches.

**Table 20.2** Command-line switches that ObjectComponents recognizes

| Switch | What the server should do | Switch placed by OLE? |
|---|---|---|
| -RegServer | Register all its information and quit. | No |
| -UnregServer | Remove all its entries from the system and quit. | No |
| -NoRegValidate | Run without confirming entries in the system database | No |
| -Automation | Register itself as single-use (one client only). Always accompanied by -Embedding. | Yes |
| -Embedding | Consider remaining hidden because it is running for a client, not for itself. | Yes |
| -Language | Set the language for registration and type libraries | No |
| -TypeLib | Create and register a type library | No |
| -Debug | Enter a debugging session | Yes |

OLE places some of the switches on the program's command line. Anyone can set other flags to make ObjectComponents perform specific tasks. An install program, for example, might invoke the application it installs and pass it the -RegServer switch to make the server register itself. Switches can begin with either a hyphen (-) or a slash (/).

Only a few of the switches call for any action from you. If a server or an automation object sees the -Embedding or -Automation switch, it might decide to keep its main window hidden. Usually ObjectComponents makes that decision for you. You can use the -Debug switch as a signal to turn trace messages on and off, but responding to -Debug is always optional. (OLE uses -Debug switch only if you register the *debugprogid* key.)

ObjectComponents handles all the other switches for you. If the user calls a program with the -UnregServer switch, ObjectComponents examines its registration tables and

erases all its entries from the registration database. If ObjectComponents finds a series of switches on the command line, it processes them all. This example makes ObjectComponents generate a type libary in the default language and then again in Belgian French.

```
myapp -TypeLib -Language=80C -TypeLib
```

The number passed to -Language must be hexadecimal digits. The Win32 API defines *80C* as the locale ID for the Belgian dialect of the French language. For this command line to have the desired effect, of course, *myapp* must supply Belgian French strings in its XLAT resources. For more information about localization, see "Localizing symbol names" on page 397.

The -RegServer flag optionally accepts a file name.

```
myapp -RegServer = MYAPP.REG
```

This causes ObjectComponents to create a registration data file in MYAPP.REG. The new file contains all the application's registration data. If you distribute MYAPP.REG with your program, users can merge the file directly into their own registration database (using RegEdit). Without a file name, -RegServer writes all data directly to the system's registration database.

For a description of the -TypeLib switch, see "Creating a type library" on page 405.

**Note** Only EXE servers have true command lines. OLE can't pass command line switches to a DLL . ObjectComponents simulates passing a command line to a DLL server so that you can use the same code either way. The registrar object always sets the right running mode flags. For more about DLL servers, see "Making a DLL server" on page 374.

## 3. Drawing, loading, and saving objects

A server must coordinate with its client to process its objects when they need to be painted or saved.

### Telling clients when an object changes

Whenever the server makes any changes that alter the appearance of an object, the server must tell OLE. OLE keeps a metafile representation with every linked or embedded object so that even when the server is not active OLE can still draw the object for the container. If the object changes, OLE must update the metafile. The server notifies OLE of the change by calling *TOleView::InvalidatePart*. OLE, in turn, asks the server to paint the revised object into a new metafile. ObjectComponents handles this request by passing the metafile device context to the server's *Paint* procedure. You don't need to write extra code for updating the metafile.

A good place to call *InvalidatePart* is in the handlers for the messages that ObjectWindows sends to a view when its data changes:

```
bool TDrawView::VnRevert(bool /*clear*/) {
  Invalidate();              // force full repaint
  InvalidatePart(invView);   // tell container about the change
  return true;
}
```

*invView* is an enumeration value, defined by ObjectComponents, indicating that the view is invalid and needs repainting.

Other view notification messages that signal the need for an update include EV_VN_APPEND, EV_VN_MODIFY, and EV_VN_DELETE.

### Loading and saving the server's documents

When a server gives objects to containers, the containers assume the burden of storing the objects in files and reading them back when necessary. If your server can also run independently and load and save its own documents, it too should make use of the compound file capabilities built into *TOleDocument*. For more about compound files, see "Loading and saving compound documents" on page 312.

In its *Open* method, a server calls *TOleDocument::Open*. In its *Commit* method, a server should call *TOleDocument::Commit* and *TOleDocument::CommitTransactedStorage*.

```
// document class declaration derived from TOleDocument
class _DOCVIEWCLASS TMyDocument : public TOleDocument {
  // declarations
}

// document class implementation
bool TMyDocument::Commit(bool force) {
  TOleDocument::Commit(force);   // save linked and embedded objects
  ⋮                              // code to save other document data
  TOleDocument::CommitTransactedStorage();   // write to file if transacted mode
}

bool TDrawDocument::Open(int, const char far* path) {
  TOleDocument::Open();     // load linked or embedded objects
  ⋮                         // code to load other document data
}
```

**Note** By default, *TOleDocument* opens compound files in transacted mode. Transacted mode saves changes in temporary storages until you call *CommitTransactedStorage*.

## 4. Building the server

To build the server application, include the OLE headers and link with the OLE libraries.

### Including OLE headers

The headers for a server are the same as the headers for a container. A server that uses the Doc/View model and an MDI frame window needs the following headers:

```
#include <owl/oledoc.h>      // replaces docview.h
#include <owl/oleview.h>     // replaces docview.h
#include <owl/olemdifr.h>    // replaces mdi.h
```

An SDI application includes oleframe.h instead of olemdifr.h.

### Compiling and linking

Linking and embedding servers that use ObjectComponents and ObjectWindows require the large memory model. Link them with the OLE and ObjectComponents libraries.

The integrated development environment (IDE) chooses the right build options when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OWLOCFMK.GEN file found in the EXAMPLES subdirectory.

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj myprogram.obj
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJEXE hold the name of the file to build and the names of the object files to build it from. The last line includes the OWLOCFMK.GEN file. Name your file MAKEFILE and type this at the command-line prompt:

```
make MODEL=l
```

MAKE, using instructions in OCFMAKE.GEN, will build a new makefile tailored to your project. The new makefile is called WIN16L*xx*.MAK.

For more information about OCFMAKE.GEN and the libraries needed for an ObjectComponents program, see page 314.

**Note** The first time the server runs, the registrar object records its information in the registration database. Be sure to run the server once before trying to use it with a container.

## Turning an ObjectWindows application into an OLE server

Turning a non-Doc/View ObjectWindows container into an OLE server requires a few modifications. This list describes them briefly. The sections that follow give more detail for each one.

1. Register the server.
   - Create an application dictionary object.
   - Create the server's registration tables.
   - Link the document registration structures in a list.
   - Create the registrar and call its *Run* method.
2. Set up the client window.
3. Modify the application class.
   - Derive it from *TOcModule* as well as *TApplication.*
   - Add the *CreateOleObject* method.
4. Build the application.

The following section expands on each step required to convert your ObjectWindows container into an OLE server. The code excerpts are from the OWLOCF2.CPP sample in

the EXAMPLES/OWL/TUTORIAL/OLE directory. OWLOCF2 converts the OWLOCF1 sample from a container to a server.

# 1. Registering the server

To register the server you describe it in registration tables—one table for the application and one for each type of document it creates. The document tables are put in a linked list, and the registrar object processes the information in all the tables. The registrar also needs an application dictionary object.

## Creating an application dictionary

An application dictionary tracks information for the currently active process. It is particularly useful for DLLs. When several processes use a DLL concurrently, the DLL must maintain multiple copies of the global, static, and dynamic variables that represent its current state in each process. For example, the DLL version of ObjectWindows maintains a dictionary that allows it to retrieve the *TApplication* corresponding to the currently active client process. If you convert an executable server to a DLL server, it must also maintain a dictionary of the *TApplication* objects representing each of its container clients.

The DEFINE_APP_DICTIONARY macro provides a simple and unified way to create the dictionary object for any type of application, whether it is a container, a server, a DLL, or an EXE. Insert this statement with your other static variables:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

For any application linked to the static version of the DLL library, the macro simply creates a reference to the application dictionary in ObjectWindows. For DLL servers using the DLL version of ObjectWindows, however, it creates an instance of the *TAppDictionary* class.

It is important to name your dictionary object *AppDictionary* to take advantage of the factory templates such as *TOleFactory* (as explained in the section "Creating the registrar object").

## Creating registration tables

Servers implement OLE objects that any container can link or embed in their own documents. Different servers implement different types of objects. Every type of object a server can create must have a 16-byte globally unique identifier (GUID) and a unique string identifier. Every server must record this information, along with other descriptive information, in the registration database of the system where it runs. OLE reads the registry to determine what objects are available, what their capabilities are, and how to invoke the application that creates objects of each type.

A server provides registration information to ObjectComponents using macros to build registration tables: one table describing the application itself and one for each type of OLE object the server creates. Here is the application registration table from OWLOCF2.

```
REGISTRATION_FORMAT_BUFFER(100)

// application registration table
```

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,      "{B6B58B70-B9C3-101B-B3FF-86C8A0834EDE}")
  REGDATA(description,"Scribble Pad Server")
END_REGISTRATION
```

The registration macros build a structure of items. Each item contains a key, such as *clsid* or *description*, and a value assigned to the key. The order in which the keys appear does not matter. In the example, *AppReg* is the name of the structure that holds the information in this table.

Servers that create several types of objects must build a document registration table for each type. (What the server creates as a document is presented through OLE as an object.) If a spreadsheet application, for example, creates spreadsheet files and graph files, and if both kinds of documents can be linked or embedded, then the application registers two document types and creates two document registration tables.

The OWLOCF2 sample program creates one type of object, a scribbling pad, so it requires one document registration table (shown here) in addition to the application registration table.

```
// document registration table
BEGIN_REGISTRATION(DocReg)
  REGDATA(progid,      "Scribble.Document.3")
  REGDATA(description,"Scribble Pad Document")
  REGDATA(debugger,    "tdw")
  REGDATA(debugprogid,"Scribble.Document.3.D")
  REGDATA(debugdesc,   "Scribble Pad Document (debug)")
  REGDATA(menuname,    "Scribble")
  REGDATA(insertable,  "")
  REGDATA(extension,   DocExt)
  REGDATA(docfilter,   "*."DocExt)
  REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGDATA(verb0,       "&Edit")
  REGDATA(verb1,       "&Open")
  REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

The *progid* key is an identifier for this document type. The string must be unique so that OLE can distinguish one object from another. The *insertable* key indicates that this type of document should be listed in the Insert Object dialog box (see Figure 18.2 on page 269). The *description, menuname,* and *verb* keys are all visible to the user during OLE operations. The *description* appears in the Insert Object dialog box where the user sees a list of objects available in the system. The *menuname* is used in the container's Edit menu when composing the string that pops up the verb menu, which is where the *verb* strings appear. The remaining registration items are used when the application opens a file or uses the Clipboard.

For a list of the keys a server should register, see Table 20.1. For more information about particular keys, see the *ObjectWindows Reference Guide*.

Place your registration structures in the source code file where you implement your *TApplication*-derived class. If you cut and paste registration tables from other programs, be sure to modify at least the *progid* and *clsid* because these must identify your

application uniquely. (Use the GUIDGEN.EXE utility to generate new 16-byte *clsid* identifiers.)

## Creating the document list

The registration tables hold information about your application and its documents, but they are static. They don't do anything with that information. To register the information with the system, an application must pass the structures to an object that know how to use them. That object is the registrar, which records any necessary information in the system registration database.

In a Doc/View application, the registrar examines the list of document templates to find each document registration structure. A non-Doc/View application doesn't have document templates, so it uses *TRegLink* instead to create a linked list of all its document registration tables.

```
static TRegLink* RegLinkHead;
TRegLink scribbleLink(::DocReg, RegLinkHead);
```

*RegLinkHead* points to the first node of the linked list. *scribbleLink* is a node in the linked list. The *TRegLink* constructor follows *RegLinkHead* to the end of the list and appends the new node. Each node contains a pointer to a document registration structure. In OWLOCF2, the list contains only one node because the server creates only one type of document. The node points to *DocReg*.

OWLOCF2 declares *RegLinkHead* as a static variable because it is used in several parts of the code, as the following sections explain.

## Creating the registrar object

The registrar object registers and runs the application. Its constructor receives the application registration structure and a pointer to the list of document registration structures. In a linking and embedding application, the registrar is an object of type *TOcRegistrar*. At the top of your source code file, declare a global variable holding a pointer to the registrar.

```
static TPointer<TOcRegistrar> Registrar;
```

The *TPointer* template ensures that the *TOcRegistrar* instance is deleted when the program ends.

**Note** Name this variable *Registrar* to take advantage of the factory callback template used in the registrar's constructor.

Next, in *OwlMain* allocate a new *TOcRegistrar* object and initialize the global pointer *Registrar*. The *TOcRegistrar* constructor has three required parameters: the application's registration structure, the component's factory callback and the command-line string that invoked that application.

An optional fourth parameter points to the beginning of the document registration list. In a Doc/View application, this parameter defaults to the application's list of document templates. Applications that do not use Doc/View should pass a *TRegLink** pointing to the list of document registration structures.

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    // construct a registrar object to register the application
    Registrar = new TOcRegistrar(::AppReg,      // application registration structure
                TOleFactory<TScribbleApp>(),    // factory callback
                TApplication::GetCmdLine(),     // app's command line
                ::RegLinkHead);                 // pointer to doc registration structures

   // did command line say to register only?
   if (Registrar->IsOptionSet(amAnyRegOption))
       return 0;
    return Registrar->Run();                    // enter message loop in factory callback
  }
  catch (xmsg& x) {
    ::MessageBox(0, x.why().c_str(), "Scribble App Exception", MB_OK);
  }
  return -1;
}
```

*TOleFactory* is a template that creates a class with a factory callback function. For a linking and embedding ObjectWindows application that does not use Doc/View, the template is called *TOleFactory*. The code in the factory template assumes you have defined an application dictionary called *AppDictionary* and a *TOcRegistrar** called *Registrar*.

When the registrar is created, it compares the information in the registration tables to the application's entries in the system registration database and updates the database if necessary. The *Run* method causes the registrar to call the factory callback which, among other things, enters the application's message loop.

For information about what the register looks for on the command line that you pass it, see "Processing the command line" on page 349.

## 2. Setting up the client window

ObjectComponents applications need to have a separate window for each document. The document window derives from *TOleFrame* and usually is made to fill the client area of a frame window. For a explanation of how the OWLOCF sample program uses a client window, see "3. Setting up the client window" on page 319. Follow the instructions there to add a client window if your application doesn't have one already.

### Creating helper objects for a document

Each new document you open needs two helper objects from ObjectComponents: *TOcDocument* and *TOcView*. Because you create a client window for each document, the window's constructor is a good place to create the helpers. The *TOleWindow::CreateOcView* function creates both at once.

In OWLOCF2, the client window is *TScribbleWindow*. Here is the declaration for the class and its constructor:

```
class TScribbleWindow : public TOleWindow {
  public:
    TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData& fileData);
    TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData& fileData, TRegLink* link);
```

The second constructor is new. It is useful when ObjectComponents passes you a pointer to the registration information you provided for one of your document types and asks you to create a document of that type. Here is the implementation of the new constructor:

```
TScribbleWindow::TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData& fileData,
    TRegLink* link)
:
  TOleWindow(parent, 0),
  FileData(fileData)
{
  ⋮
  // Create a TOcDocument object to hold the OLE parts that we create
  // and a TOcRemView to provide OLE services
  CreateOcView(link, true, 0);
}
```

The constructor receives a *TRegLink* pointer and passes it on to *CreateOcView*. The pointer points to the document registration information for the type of document being created. ObjectComponents passes the pointer to this constructor; you don't have to keep track of it yourself.

Passing **true** to *CreateOcView* causes the function to create a *TOcRemView* helper instead of a *TOcView*. The remote view object draws an OLE object within a container's window. When a server is launched to help a client with a linked or embedded object, it should create a remote view.

If your application supports more than one document type, you can choose to use a different *TOleWindow*-derived class for each one. You must then provide the additional constructor for each class. Alternatively, you can use a single *TOleWindow*-derived class that behaves differently depending on the *TRegList* pointer it receives.

## 3. Modifying the application class

ObjectComponents requires that the class you derive from *TApplication* must also inherit from *TOcModule*, as described in "Deriving the application object from TOcModule" on page 343. In addition, the application object needs to implement a *CreateOleObject* method with the following signature:

```
TUnknown* CreateOleObject(uint32 options, TRegList* link);
```

The purpose of the function is to create a server document for linking or embedding. The server must create a client window and return a pointer of type *TOcRemView**. Here is how OWLOCF2 declares this procedure:

```
class TScribbleApp : public TApplication, public TOcModule {
  public:
    TScribbleApp();
    TUnknown* CreateOleObject(uint32 options, TRegLink* link);
```

And here is how it implements the procedure:

```
TUnknown*
TScribbleApp::CreateOleObject(uint32 options, TRegLink* link)
{
  if (!link)  // factory creating an application only, no view required
    link = &scribbleLink;       // need to have a view for this app
  TOleFrame* olefr = TYPESAFE_DOWNCAST(GetMainWindow(), TOleFrame);
  CHECK(olefr);
  FileData.FileName[0] = 0;
  TScribbleWindow* client = new TScribbleWindow(
          olefr->GetRemViewBucket(), FileData, link);
  client->Create();
  return client->GetOcRemView();
}
```

ObjectWindows uses the *CreateOleObject* method to inform your application when OLE needs the server to create an object. The *TRegLink** parameter indicates which object to create.

## Understanding the TRegLink document list

This section explains the relationship between the document registration structure, the document list, and the *CreateOleObject* method. A server builds a document registration table for each type of object that it can serve. (The variable that holds the document registration information is conventionally named *DocReg*.) The registration structure is then passed to a *TRegLink* constructor, which appends the the structure to a linked list so that all the document types can be registered.

OLE displays the *description* value for each document in the Insert Object dialog box whenever the user asks to insert an object. (OLE also displays the *description* strings for all the other available server document types.) When the user chooses to insert one of your objects into a container application, OLE launches your server and places the -Embedding switch on the command line. When the server loads, ObjectComponents calls your *CreateOleObject* method, passing the address of the registration link that was used to register the requested document type. The *TRegList* pointer lets you determine which type of object was chosen. This matters primarily for servers that register more than one document type.

The following code illustrates one possible implementation of the *CreateOleObject* method for an application that serves more than one type of object:

```
// Create a appropriate client window and return its TOcRemView pointer
TUnknown*
TServerApp::CreateOleObject(uint32 options, TRegList* link)
{
  if (link == &chartLink) {
    // Create TOleChartWindow
    // and return charWindow->GetOcRemView();
  }

  if (tpl == &worksheetLink) {
    // Create TOleWorksheetWindow
```

```
    // and return worksheetWindow->GetOcRemView();
  }

  return 0;
}
```

## 4. Building the server

To build the server application, include the OLE headers and link with the OLE libraries.

### Including OLE headers

ObjectWindows provides OLE-related classes, structures, macros, and symbols in various header files. The following list shows the headers needed for an OLE container using an SDI frame window.

```
#include <owl/oleframe.h>
#include <owl/olewindo.h>
#include <ocf/ocstorag.h>
```

An MDI application includes olemdifr.h instead of oleframe.h.

### Compiling and linking

Linking and embedding servers that use ObjectComponents and ObjectWindows must be compiled with the large memory model. They must be linked with the OLE and ObjectComponents libraries. For help building a makefile, see "Compiling and linking" on page 352.

**Note** The first time the server runs, the registrar object records its information in the registration database. Be sure to run the server once before trying to use it with a container.

# Turning a C++ application into an OLE server

If you are writing a new program, consider using ObjectWindows to save yourself some work. The ObjectWindows Library contains built-in code that automatically performs some tasks common to all ObjectComponents programs. Programs that don't use ObjectWindows must undertake these chores for themselves.

The following list briefly describes what you need to do to turn a C++ application into an ObjectComponents server. Many of the items in this list also appear on page 328 in the list of changes for creating a container. If you have already turned your C++ application into a container, much of the server work is already done. The most important differences concern the registration tables and the factory callback function.

The sections that follow explain each step in more detail.

1 Create a memory allocator object in *WinMain*.

2 Register the application.

- Build registration tables.
- Create a document list.
- Create a registrar object and call its *Run* function.
- Write a factory callback function to create the application or its documents on demand.

3 Create a view window to display an open document.

- Create, resize, and destroy the view window together with the main window.
- Create supporting objects for a new server document.
- Make the view window handle the WM_OCEVENT message.
- Write handlers for selected ObjectComponents view events.
- Paint the server's document in the view window.

4 Program the main window to handle OLE commands and events.

- When creating the main window, call *TOcApp::SetupWindow*.
- Make the main window handle the WM_OCEVENT message.
- Write handlers for selected ObjectComponents application events.

5 Build the application.

- Include ObjectComponents headers.
- Link to the ObjectComponents and OLE libraries.

The sections that follow illustrate each step using examples from the programs in the EXAMPLES/OCF/CPPOCF directory. The source files titled CPPOCF0 contain a Windows application that does not support OLE. CPPOCF1 turns the first program into an OLE container. CPPOCF2 adds server capabilities. The code samples for this discussion come from CPPOCF2. The CPPOCF2 server creates a simple timer display for containers to embed. The timer display increments every second.

This chapter does not describe all the changes between CPPOCF1 and CPPOCF2, only those that pertain to OLE features.

## 1. Creating a memory allocator

A server adds this line to the beginning of its *WinMain* procedure:

```
TOleAllocator allocator(0);      // use default memory allocator
```

The allocator's constructor initializes the OLE libraries and its destructor releases them when the object goes out of scope. Passing 0 to the constructor tells it to let OLE use its standard memory functions whenever allocating memory for this application.

## 2. Registering the application

CPPOCF2 supports basic server functions. It registers information about itself and its document type, and it creates on demand an object to embed in other applications.

## Building registration tables

A server uses the registration macros to build registration tables describing the application and the documents it creates. The first table describes the server itself:

```
REGISTRATION_FORMAT_BUFFER(100)
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,        "{BD5E4A81-A4EF-101B-B31B-0694B5E75735}")
  REGDATA(description,  "Sample C Server")
END_REGISTRATION
```

The registration macros build a structure of type *TRegList*. The structure is stored in a variable named *AppReg*. Each entry in the structure contains a key, such as *clsid* or *description*, and a value assigned to the key. Internally, ObjectComponents finds the values by searching for the keys. The order in which the keys appear does not matter.

The server creates a second registration table to describe the type of document it produces. If a spreadsheet application, for example, creates spreadsheet files and graph files, it registers two document types. CPPOCF2 creates only one kind of document, a timer display. The registration structure for this document type is held in a variable named *DocReg*, as the following code shows.

```
BEGIN_REGISTRATION(DocReg)
  REGDATA(description,  "Sample C Server Document")
  REGDATA(progid,       APPSTRING".Document.1")
  REGDATA(menuname,     "CServer")
  REGDATA(insertable,   "")
  REGDATA(verb0,        "&Edit")
  REGDATA(verb1,        "&Open")
  REGDATA(extension,    "scd")
  REGDATA(docfilter,    "*.scd")
  REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

The *progid* key is an identifier for this document type. The *insertable* key indicates that this type of document should be listed in the Insert Object dialog box (see Figure 18.2 on page 269). The *description*, *menuname*, and *verb* keys are all visible to the user during OLE operations. The *description* appears in the Insert Object dialog box. The *menuname* is used in the container's Edit menu when composing the string that pops up the verb menu, which is where the *verb* strings appear.

The remaining registration items are used when the application opens a file or uses the clipboard. For a list of keys a server should register, see Table 20.1. For descriptions of individual keys, see the *ObjectWindows Reference Guide*.

## Creating the document list

The registration tables hold information about your application and its documents, but they are static. They don't do anything with that information. To register the information in the system, an application must pass the structures to objects that know how to use them. That object is the registrar, which records any necessary information in the system registration database.

To accommodate servers with many document types, the registrar accepts a pointer to a linked list of all the application's document registration structures. Each node in the list is a *TRegLink* object. Each node contains a pointer to one document registration structure and another pointer to the next node.

```
TRegLink *RegLinkHead = 0;
TRegLink  regDoc(DocReg, RegLinkHead);
```

*RegLinkHead* points to the first node of the linked list. *regDoc* is a node in the linked list. The *TRegLink* constructor follows *RegLinkHead* to the end of the list and appends the new node. Each node contains a pointer to a document registration structure. In CPPOCF2, the list contains only one node because the server creates only one type of document. The node points to *DocReg*.

CPPOCF2 declares *RegLinkHead* as a static variable because it is used in several parts of the code, as the following sections explain.

## Creating the registrar object

The registrar object records application information in the system registration database, processes any OLE switches on the application's command line, and notifies OLE that the server is running. CPPOCF2 declares a static pointer for the registrar object:

```
TOcRegistrar* OcRegistrar = 0;
```

Create your registrar object as you initialize the application in *WinMain*. Instead of entering a message loop, call the registrar's *Run* method. When *Run* returns, the application is shutting down. Delete the registrar before you quit. The CPPOCF2 *WinMain* function shows all the steps.

```
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, char far* lpCmdLine, int nCmdShow)
{
  HInstance = hInstance;
  try {
    TOleAllocator allocator(0);     // use default memory allocator

    string cmdLine(lpCmdLine);      // put app's command line in a C++ string

    // construct the registrar
    OcRegistrar = new TOcRegistrar(::AppReg,     // application registration structure
          ComponentFactory,                      // factory callback function
          cmdLine,                               // application command line
          ::RegLinkHead);                        // document registration structure list

    :   // application initialization commands go here

    if (OcRegistrar->IsOptionSet(amEmbedding))
      nCmdShow = SW_HIDE;

    :   // instance initialization commands go here

    OcRegistrar->Run();             // call factory asking app to create itself and run
    delete OcRegistrar;
  }
```

```
    catch (xmsg& x) {
      MessageBox(GetFocus(), x.why().c_str(), "Exception caught", MB_OK);
    }
    return 0;
  }
```

The *TOcRegistrar* constructor takes four parameters:

- *::AppReg*, is the application registration structure already built with the registration macros (see the section "Building registration tables").
- *ComponentFactory* is a callback function and is described in the next section. The callback is responsible for creating any of the application's OLE components, including the application itself, as required. The callback also contains the application's message loop.
- *cmdLine* is a *string* object holding the application's command line. The registrar searches the command line for OLE-related switches such as -Automation or -Embedding, and it sets internal running mode flags accordingly.
- *::RegLinkHead* points to the linked list of documentation registration structures. (See the preceding section "Creating the document list.")

During its initialization, the server checks whether it was invoked by OLE or directly by the user. OLE launches the server when the user activates a linked or embedded object that the server created. OLE sets the -Embedding switch on the server's command line to indicate that the server is running only to support a client, not as a stand-alone application. When the registrar discovers the -Embedding switch on the command line, it sets an internal flag. The server tests for this flag by calling *IsOptionSet*. If OLE did launch the application, the server will draw only in the container's window and it does not need to display its own window. CPPOCF2 sets *nCmdShow* to SW_HIDE to prevent subsequent initialization code from displaying the main window.

*nCmdShow* is the parameter Windows passes to *WinMain* indicating the initial state of the application's main window. A well behaved Windows application passes the value to *ShowWindow* immediately after creating the main window.

The *TOcRegistrar::Run* function causes the registrar to call the application's factory callback. In this case, the callback executes the application's message loop and the application runs.

## Writing the factory callback function

The factory callback is a function you implement and pass to the registrar. When it is time for the application to run, or when a container tries to insert one of the server's objects, ObjectComponents invokes the callback function.

The factory callback decides what to do by reading the parameters it receives and examining the running mode flags the registrar has set. The callback is called a factory because it creates OLE component objects on request.

The requirement that every ObjectComponents application must supply a factory callback function unifies the process of creating objects. Normally the process varies depending on whether the application is a container or a server, whether it is automated, whether it is running as a DLL or an executable program, and whether the

application was invoked by the user directly or by OLE. The factory callback makes it possible to revise and run the application in a variety of ways without rewriting any code. For more information about factory callbacks, look up "Factory Templates" in the *ObjectWindows Reference Guide.*

A set of factory templates such as *TOleFactory* and *TOleAutoFactory* make it easy to implement factories for ObjectWindows programs, but in a straight C++ program you have to write the factory yourself.

Factory callback procedures can have any name you like, but they must follow this prototype:

```
IUnknown* ComponentFactory(IUnknown* outer, uint32 options, uint32 id);
```

*outer* is used when aggregating OLE objects to make them function as a single unit. The factory's return value is also used for aggregation. Because containers don't aggregate, CPPOCF1 ignores *outer* and returns 0.

*options* contains the bit flags that indicate the application's running mode. The registrar object sets the flags when it processes the command line switches, before it calls the factory callback. The factory tests the flags to find out what it should do. The possible flags are defined by the *TOcAppMode* enumerated type, and they have names like *amRun* and *amShutdown.*

*id* is an identifier that tells the factory what kind of object to create.

The factory's parameters can direct the factory to perform one of three actions:

- Initialize the application. The first time it runs, the factory creates a *TOcModule* object. *TOcModule* connects the application to the OLE system by creating a *TOcApp* connector object. The factory also handles aggregation in this phase.
- Run the application. If the *amRun* flag is set, the factory enters the message loop. If the server is built as a DLL, then when OLE loads the server the registrar does not set the *amRun* flag and the server should not run its own message loop.
- Create an object. The *id* parameter tells the factory what kind of object to create. Because CPPOCF2 creates only one kind of object, it checks only whether *id* is greater than 0. In applications that register multiple document templates, *id* points to the template for the requested object.

The factory callback in CPPOCF2 refers to four global variables. One is *OcRegistrar,* explained earlier. Another is *OcApp.*

```
TOcRegistrar* OcRegistrar = 0;
TOcApp*       OcApp       = 0;
```

*TOcApp* is the connector object that implements OLE interfaces on behalf of the application. One of the factory's jobs is to create the connector object when the application starts and to destroy it when the application shuts down.

The other two global variables, *OcDoc* and *OcRemView,* are explained in the next section.

Here is the factory callback from CPPOCF2:

```
IUnknown*
ComponentFactory(IUnknown* outer, uint32 options, uint32 id)
```

```
{
  IUnknown* ifc = 0;

  // start the application or shut it down
  if (!OcApp) {
    if (options & amShutdown)  // no app to shutdown!
      return 0;
    OcRegistrar->CreateOcApp(options, OcApp);
  } else if (options & amShutdown) {
    DestroyWindow(HwndMain);
    return 0;
  }

  // aggregate if an outer pointer was passed
  if (id == 0)
    OcApp->SetOuter(outer);

  // enter message loop if the run flag is set
  if (options & amRun) {
    if ((options & amEmbedding) == 0) {
      HwndView = CreateViewWindow(HwndMain);
    }
    MSG msg;
    // Standard Windows message loop
    while (GetMessage(&msg, 0, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }

  // create a document if the id parameter is non-zero
  if (id) {
    OcDoc     = new TOcDocument(*OcApp);
    HwndView  = CreateViewWindow(HwndMain);
    OcRemView = new TOcRemView(*OcDoc, &DocReg);
    if (IsWindow(HwndView))
      OcRemView->SetupWindow(HwndView);
    ifc = OcRemView->SetOuter(outer);
  }
  return ifc;
}
```

The factory's *outer* parameter is 0 unless some other object is aggregating with the newly created object. Aggregated objects are components that act together as a single unit. Objects can form aggregations at run time; you do not need access to an object's source code to aggregate with it. ObjectComponents supports aggregation by passing *outer* to the application factory. If *outer* is non-zero, it points to the *IUnknown* interface of anther object that wants the newly created object to subordinate itself. To allow aggregation, the factory calls the *SetOuter* method on the object it is creating, either *TOcApp* or *TOcRemView*. *SetOuter* returns a pointer to the object's own *IUnknown* interface. The factory should return the same pointer, too.

**Note** *TOcApp::SetOuter* is only called when an application automates itself. CPPOCF2 includes the call anyway in case the application later becomes an automation server.

## 3. Creating a view window

ObjectComponents imposes one design requirement on servers: the server document must have its own window, separate from the application's main window. To keep the distinction clear, we'll call the main window the *frame* window, because it uses the WS_THICKFRAME style and has a visible border on the screen. The second window has no visible border. We'll call it the *view* window because that is where the application displays its data. The view window always exactly fills the frame window's client area, so from the user's point of view the frame window appears to be the only window. ObjectComponents needs the view window, though, because it expects to send some event messages to the application and some to the view.

In an SDI application like the CPPOCF2 sample program, the frame window controls the view window. When the frame window receivesa WM_SIZE message, it moves the view to keep it aligned with the frame's client area. When it receives WM_CLOSE, it destroys both itself and the view window.

In an MDI application, each child window creates its own view. The child window does what the SDI frame does: creates and manages a view for the document it displays.

### Creating, resizing, and destroying the view window

Before creating the view window, the application must first register a class for the view window. CPPOCF2 registers both classes in *InitApplication*.

CPPOCF2 creates the view window in its factory because the factory is in charge of creating new documents on request. The code for the view window, as you'll see, connects the new document to OLE by creating some ObjectComponents helper objects. The factory calls this function to create the view window:

```
HWND CreateViewWindow(HWND hwndParent)
{
  HWND hwnd = CreateWindow(VIEWCLASSNAME, "",
    WS_CHILD | WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_VISIBLE | WS_BORDER,
    10, 10, 300, 300,
    hwndParent, (HMENU)1, HInstance, 0);
  return hwnd;
}
```

CPPOCF2 resizes and destroys the view window when the frame window receives WM_SIZE and WM_CLOSE messages.

```
void
MainWnd_OnSize(HWND hwnd, UINT /*state*/, int /*cx*/, int /*cy*/)
{
  if (IsWindow(HwndView)) {
    TRect rect;
    GetClientRect(hwnd, &rect);
    MoveWindow(HwndView, rect.left, rect.top, rect.right, rect.bottom, true);
  }
```

```
}

void
MainWnd_OnClose(HWND hwnd)
{
  if (IsWindow(HwndView))
    DestroyWindow(HwndView);
  DestroyWindow(hwnd);
}
```

The view window always fills the frame window's client area exactly. If the user opens and closes documents or embeds objects, the changes show up in the view window.

## Creating a new server document

For every open document, the server needs to create two helper objects: *TOcDocument* and *TOcRemView*. The document helper manages the collection of objects inserted in the document. (It is possible for objects to be embedded within objects.) The view helper connects the document to OLE. More specifically, it implements interfaces that OLE can call to communicate with the document. When OLE tells the view object that something noteworthy has occurred, the view object sends a message to the view window. (The next two sections show how to handle the messages.)

The sample CPPOCF2 server creates the two helpers in its factory when asked to create a new document.

```
OcDoc      = new TOcDocument(*OcApp);
HwndView   = CreateViewWindow(HwndMain);
OcRemView = new TOcRemView(*OcDoc, &DocReg);
if (IsWindow(HwndView))
  OcRemView->SetupWindow(HwndView);
```

The *SetupWindow* method tells the *TOcRemView* object where to send event messages. In this case, it sends messages to *HwndView*, the view window. The view window now receives WM_OCEVENT messages.

For each new document a server creates a *TOcDocument*, a *TOcRemView*, and a view window. The same objects are deleted or released when the view window is destroyed:

```
void
ViewWnd_OnDestroy(HWND hwnd)
{
  ⋮                                // other document cleanup can go here

  if (OcRemView)
    OcRemView->ReleaseObject();    // do not delete a TOcRemView object

  if (OcDoc) {
    OcDoc->Close();                // release the servers for any embedded parts
    delete OcDoc;                  // this is not a COM object, so you can delete it
  }

  if (IsWindow(HwndMain))
    PostMessage(HwndMain, WM_CLOSE, 0, 0);
```

```
    HwndView = 0;
  }
```

When the view window is destroyed, it makes three calls to dispose of the helper objects. *OcRemView->ReleaseObject* signals that the view window is through with the *TOcRemView* connector object. You shouldn't call **delete** for a view object because the OLE system might still need more information before it allows the view to shut down. *ReleaseObject* decrements an internal reference count and dissociates the view from its window. When all the clients of the view object have released it, the count reaches 0 and the object destroys itself.

The *TOcDocument* view object, on the other hand, is not a connector object and so you can destroy it with **delete** in the usual way. First, however, you should call *Close* to release the server applications that OLE may have invoked to support objects embedded in the server's document.

Because CPPOCF2 never opens more than one document at a time, it declares *OcDoc* and *OcRemView* as static global pointers.

```
TOcDocument*  OcDoc      = 0;
TOcRemView*   OcRemView  = 0;
```

A server that supports concurrent clients with a single instance of the application, or one that uses the multidocument interface (MDI), needs to create a different *TOcDocument* and *TOcRemView* pair for each document window.

**Note** When launched to support an object in a container, servers create *TOcRemView* instead of *TOcView* because they are painting in a remote window. For simplicity, CPPOCF2 always creates a remote view even when it is launched directly by the user. The only penalty is extra overhead.

## Handling WM_OCEVENT

Because the *TOcRemView::SetupWindow* method bound the *OcRemView* connector to the view window, the connector sends its event notification messages to the window. All ObjectComponents events are sent in the WM_OCEVENT message, so the view window procedure must respond to WM_OCEVENT.

```
long CALLBACK _export
ViewWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    :  // other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  ViewWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}
```

The HANDLE_MSG message cracker macro for WM_OCEVENT is defined in the ocf/ocfevx.h header. The same header also defines a another cracker for use in the WM_OCEVENT message handler.

```
// Subdispatch OC_VIEWxxxx messages
long
ViewWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
```

```
{
  switch (wParam) {
    // insert an event cracker for each OC_VIEWxxxx message you want to handle
    HANDLE_OCF(hwnd, OC_VIEWCLOSE, ViewWnd_OnOcViewClose);
  }
  return true;
}
```

The WM_OCEVENT message carries an event ID in its *wParam*, just as WM_COMMAND messages carry command IDs. OC_VIEWCLOSE is one possible event, indicating that it is time to close this view. In applications that show only one view per document, OC_VIEWCLOSE also signals the close of the document. The HANDLE_OCF macro calls the handler you designate for each ObjectComponents event, just as HANDLE_MSG calls the handler for for a window message.

CPPOCF2 handles only the OC_VIEWCLOSE message. To handle others, add one HANDLE_OCF macro for each event ID. A list of all the ObjectComponents messages appears in Tables 18.5 and 18.6.

## Handling selected view events

Each HANDLE_OCF macro calls a different handler function. In the example, the handler function is called *ViewWnd_OnOcViewClose*.

```
bool
ViewWnd_OnOcViewClose(HWND hwnd)
{
  DestroyWindow(hwnd);
  return true;
}
```

A server receives this message when a container closes the document that contains the server's object. CPPOCF2 responds by closing the view window. The WM_DESTROY handler also deletes or releases the helper objects associated with the server document.

## Painting the document

No special code is required in the server's paint procedure. It always paints its document the same way, whether or not it is painting an embedded object.

```
void
ViewWnd_OnPaint(HWND hwnd)
{
  PAINTSTRUCT ps;
  HDC dc = BeginPaint(hwnd, &ps);
  wsprintf(Buffer, "%u", Counter);
  TextOut(dc, 0, 0, Buffer, lstrlen(Buffer));
  EndPaint(hwnd, &ps);
}
```

When the view window is created, it starts off a timer. Every time the view receives a WM_TIMER message, it increments the value in the global variable *Counter* and calls *InvalidateRect* to make the view repaint itself. On each call, the paint procedure prints the value of *Counter*.

## 4. Programming the main window

The view window manages tasks related to a single document. It opens and closes the document and draws it on the screen. The frame window manages tasks for the whole application. It responds to menu commands, and it creates and destroys the view window.

### Creating the main window

When the application creates its main window, it must bind the window to its *TOcApp* object. (The *TOcApp* object was created in the factory callback function.)

```
bool
MainWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  if (OcApp)
    OcApp->SetupWindow(hwnd);

  HwndMain = hwnd;
  return true;
}
```

The *TOcApp* object sends messges about OLE events to the application's main window. *SetupWindow* tells the *TOcApp* where to direct its event notifications.

### Handling WM_OCEVENT

Like the view window, the frame window also receives WM_OCEVENT messages. The frame window receives notification of events that concern the application as a whole and not just a particular document. The frame window procedure sends WM_OCEVENT messages to a handler that identifies the event and calls the appropriate handler routine. Both routines closely resemble the corresponding code for the view window.

```
// Standard message-handler routine for main window
long CALLBACK _export
MainWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    : // other message crackers go here
    HANDLE_MSG(hwnd, WM_OCEVENT,  MainWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}

// Subdispatch OC_... messages
long
MainWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    HANDLE_OCF(hwnd, OC_APPSHUTDOWN, MainWnd_OnOcViewTitle);
  }
```

```
    return true;
  }
```

### Handling selected application events

The only ObjectComponents event that CPPOCF2 can handle in its main window is OC_APPSHUTDOWN. A server receives this message when the last linked or embedded object closes down. If the server was launched by OLE, it can terminate. If user launched the server directly, the server doesn't need to do anything.

```
const char*
MainWnd_OnOcAppShutDown(HWND hwnd)
{
  if (OcRegistrar->IsOptionSet(amEmbedding))
    DestroyWindow(hwnd);
}
```

The registrar sets the *amEmbedding* flag at startup if it finds the -Embedding switch on the application's command line. OLE pass the -Embedding switch when it launches a server to support a linked or embedded object.

## 5. Building the server

To build the server, you need to include the right headers, use a supported memory model, and link to the right libraries.

### Including ObjectComponents headers

The following list shows the headers needed for an ObjectComponents server that does not use ObjectWindows.

```
#include <ocf/ocapp.h>       // TOcRegistrar, TOcModule, TOcApp (application connector)
#include <ocf/ocreg.h>       // registration constants and app mode flags
#include <ocf/ocdoc.h>       // TOcDocument (compound document manager)
#include <ocf/ocview.h>      // TOcView (document view connector)
#include <ocf/ocpart.h>      // TOcPart (linked/embedded object connector)
#include <ocf/ocremvie.h>    // TOcRemView (document remote view connector)
#include <ocf/ocfevx.h>      // WM_OCEVENT message crackers
```

### Compiling and linking

ObjectComponents applications that do not use ObjectWindows work with either the medium or large memory model. They must be linked with the OLE and ObjectComponents libraries.

To build CPPOCF0, CPPOCF1, and CPPOCF2, move to the program's directory and type this at the command prompt:

```
make MODEL=l
```

This command builds all three programs using the large memory model.

The make file that builds this example program refers to the OCFMAKE.GEN file. For more information about using OCFMAKE.GEN in your own make files, see page 359.

**Note** The first time the server runs, the registrar object records its information in the registration database. Be sure to run the server once before trying to use it with a container.

# Understanding registration

This section explains how the registration macros work.

The BEGIN_REGISTRATION and END_REGISTRATION macros declare a structure to hold registration keys and the values assigned to the keys. The macros that come in between, such as REGDATA and REGFORMAT, each insert an item in the structure. The main structure is of of type *TRegList* and each item in the structures is an entry of type *TRegItem*. Each item contains a key and a value for that key.

```
struct TRegItem {
  char* Key;            // standard registry key
  TLocaleString Value;  // value you assign to the key
};
```

The two parameters passed to REGDATA are a key and a value. The macros make it easy to add keys and values to the structure without having to manipulate *TRegList* and *TRegItem* objects directly yourself. At run time, ObjectComponents scans the tables and confirms that the information is stored accurately in the system registration database.

## Storing information in the registration database

ObjectComponents copies information from the program's registration tables to the system's registration database. The *TOcRegistrar* object takes care of this chore when it is constructed. Every time a server constructs its *TOcRegistrar* object, ObjectComponents confirms that the registration information is accurately recorded. ObjectComponents compares the program's current *progid*, *clsid*, and executable path with the values previously written in the registration database. Any discrepancy causes ObjectComponents to reregister the entire program automatically.

Windows 3.1 stores the registration database in the REG.DAT file. Windows NT puts it in the system registry, a facility managed privately by the operating system. In either location, ObjectComponents follows the standard logical structure for recording information about an OLE server. The registration tables shown in "Building registration tables" on page 344 produce these entries:

```
\CLSID
  {5E4BD321-8ABC-101B-A23B-CE4E85D07ED2} = Drawing Pad (Step15--Server)
  Insertable =
  AuxUserType =
    2 = Drawing Pad 15
  DataFormats =
    GetSet =
      4 = Link Source,1,4,1
      3 = 8,1,1025,1
      2 = 2,1,1040,1
      1 = 3,1,1056,1
```

```
      0 = Embed Source,1,8,1
    verb =
      1 = &Open,0,2
      0 = &Edit,0,2
    InprocHandler = ole2.dll
    ProgID = DrawingPad.Document.15
    LocalServer = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE %1
  \DrawingPad.Document.15 = Drawing Pad (Step15--Server)
    Insertable =
    shell =
      open =
        command = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE %1
      print =
        command = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE %1
    protocol =
      StdFileEditing =
        server = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE
        verb =
          1 = &Open
          0 = &Edit
    CLSID = {5E4BD321-8ABC-101B-A23B-CE4E85D07ED2}
```

To inspect the entries in your registration database, run RegEdit with the /v command-line option.

**Note** In 16-bit Windows, the registration database has a capacity of 64K. If the database fills past its capacity, OLE behaves unpredicatably, and it might be necessary to erase the your REG.DAT file. Then you need to reregister the OLE applications in your system. You can register or unregister any ObjectComponents server by passing the -RegServer or -UnregServer switch on its command line. Unregistering unused applications or obsolete versions is a good way to converve space in the database.

To learn more about the registration database, search for the topic "Registration Database" in the online Help file for 16-bit Windows programming (WIN31WH.HLP).

## Registering localized entries

The values assigned to registration parameters often need to be localized. The section called "Localizing symbol names" on page 397 explains how to create XLAT resources that contain translation tables for your OLE strings. Besides putting translations in your resources, you must also mark the strings so that ObjectComponents can tell which ones have localized versions.

```
REGDATA(description, "@myapp_description")
```

The @ prefix tells ObjectComponents that the string *myapp_description* is an identifier for an XLAT resource where the real description is stored in several languages. For more information about localizing OLE strings, refer to the *TLocaleString* entry in the *ObjectWindows Reference Guide*.

## Registering custom entries

databaseThe REGDATA macro associates strings with keys. The registrar scans the list of keys and, when needed, writes the associates strings in the system registration database. The standard keys such as *progid* and *clsid* correspond to standard entries in the database. If you want to register values for non-standard keys, use the databaseREGITEM macro. The first parameter for REGITEM is the complete key exactly as you would pass it to a function like *RegOpenKey*.

```
REGITEM("CLSID\\<clsid>\\Conversion\\Readable\\Main","FormatX,FormatY")
```

As the example shows, the REGITEM arguments can contain embedded parameter names enclosed in angle brackets. When ObjectComponents registers this item, it first replaces the expression <*clsid*> with whatever value the registration block has assigned to the *clsid* parameter.

For information about *RegOpenKey*, see the Help file for the Windows API.

# Making a DLL server

Typically, linking and embedding servers are stand-alone executables that can be launched directly by the user or invoked indirectly by an OLE container. You can also implement an OLE server in a DLL. A server built as a DLL is sometimes called an *in-process server* because DLL code runs in the same process as its client. The terms *EXE server* and *server application* refer specifically to a server implemented in an EXE. ObjectComponents allows you to create both EXE and DLL servers. If you are using ObjectWindows, converting from one to the other requires only two simple changes.

**Note** The discussion and instructions that follow apply to automation servers as well as linking and embedding servers.

## Pros and cons of DLL servers

The major advantage of DLL servers is performance. Because a DLL server lives in the address space of the container,it loads and responds very fast. An EXE server, on the other hand, is a separate process and requires some form of intertask communication to interact with a container. OLE serializes intertask calls and marshals the function calls with their parameters, packaging them into the proper format for the interprocess protocol. (The protocol it uses is called LRPC, for Lightweight Remote Procedure Call.) The process of serializing the drawing commands in a metafile is particularly slow, so DLL servers substantially increase the speed of creating presentation data for linked and embedded objects.

There are a few disadvantages to using a DLL server, however. While OLE supports interaction between 16-bit and 32-bit executable applications, a 16-bit Windows application cannot use a 32-bit DLL server and a Win32 application cannot use a 16-bit DLL server. Also, DLLs do not have message queues. As a result, a DLL server cannot easily perform a task in the background. ObjectWindows overcomes this limitation by running a timer so that it can still call the *IdleAction* methods of objects derived from *TApplication* or *TOleFrame*. (ObjectWindows also uses the timer for internal processes

such as command-enabling for tool bars, deleting condemned windows, and resuming thrown exceptions.)

Because a DLL server becomes part of the container's process, bugs in one can interfere with the other, making DLL servers sometimes harder to debug.

DLL servers also present user interface dilemmas. For example, when a container initiates an open edit session with a server, it doesn't matter to the user whether the server is an EXE or a DLL; the user interface for open editing is the same either way. But the lifetime of a DLL server is tied to the container that loads it. When the container quits, the server DLL is unloaded. That can cause problems if the server's user interface normally allows the user to edit serveral documents at once. If the user were to create a new document while editing an embedded object, the user might want to continue editing the new document even after the container quits, but then the server is no longer in memory. This is a particular problem for MDI servers because the MDI interface allows users to open multiple documents in a single session. Typically DLL servers do not allow multidocument editing.

Finally, DLL servers have one other disadvantage. While OLE 2 provides a compatibility layer to let OLE 2 servers interact with OLE 1 clients, the compatibility layer works only for EXE servers. A DLL server cannot support an OLE 1 client.

## Building a DLL server

Converting an ObjectWindows EXE server to a DLL server requires only a few modifications. The following list describes them briefly. The sections that follow give more detail for each one.

1. Update your document registration table.
2. Build the program with the DLL option enabled. Link to the DLL compatible ObjectComponents and ObjectWindows Libraries.

### Updating your document registration table

The document registration tables of DLL servers must contain the *serverctx* key with the string value "Inproc." This allows ObjectComponents to register your application as a DLL server with OLE. EXE servers do not need to use the *serverctx* key since ObjectComponents defaults to EXE registration.

The following code illustrates the document registration structure of a DLL server. It comes from the sample Tic Tac Toe program in EXAMPLES/OWL/OCF/TTT.

```
BEGIN_REGISTRATION(DocReg)
 REGDATA(progid,      "TicTacToeDll")
 REGDATA(description,"TicTacToe DLL")
 REGDATA(serverctx, "Inproc")
 REGDATA(menuname,    "TicTacToe Game")
 REGDATA(insertable, "")
 REGDATA(extension,  "TTT")
 REGDATA(docfilter,  "*.ttt")
 REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
 REGDATA(verb0,      "&Play")
```

```
 REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
 REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

You do not need to modify your application registration structure to convert your EXE server to a DLL server. It's a good idea, however, to use different *clsid* and *progid* values, especially if you intend to switch frequently from one type to the other. You can test for the BI_APP_DLL macro to declare a registration structure that works for both DLL and EXE servers; the macro is only defined when you are building a DLL. The following code shows a sample document registration which supplies two sets of *progid* and *clsid* values.

```
REGISTRATION_FORMAT_BUFFER(100)

// Application registration structure
BEGIN_REGISTRATION(AppReg)
#if defined(BI_APP_DLL)
 REGDATA(clsid,      "{029442B1-8BB5-101B-B3FF-04021C009402}")   // DLL clsid
 REGDATA(progid,     "TicTacToe.DllServer")                      // DLL progid
#else
 REGDATA(clsid,      "{029442C1-8BB5-101B-B3FF-04021C009402}")   // EXE clsid
 REGDATA(progid,     "TicTacToe.Application")                    // EXE progid
#endif
 REGDATA(description,"TicTacToe Application")                    // Description
END_REGISTRATION

// Document registration structure
BEGIN_REGISTRATION(DocReg)
#if defined(BI_APP_DLL)
 REGDATA(progid,     "TicTacToeDll")
 REGDATA(description,"TicTacToe DLL")
 REGDATA(serverctx, "Inproc")
#else
 REGDATA(progid,     "TicTacToe.Game.1")
 REGDATA(description,"TicTacToe Game")
 REGDATA(debugger,   "tdw")
 REGDATA(debugprogid,"TicTacToe.Game.1.D")
 REGDATA(debugdesc,  "TicTacToe Game (debug)")
#endif
 REGDATA(menuname,   "TicTacToe Game")
 REGDATA(insertable, "")
 REGDATA(extension,  "TTT")
 REGDATA(docfilter,  "*.ttt")
 REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
 REGDATA(verb0,      "&Play")
 REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
 REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

Notice that the debugger keys (*debugger, debugprogid,* and *debugdesc*) are not used when building a DLL server. They are relevant only when your server is an executable that a debugger can load. (To debug a DLL server, see "Debugging a DLL server").

### Compiling and linking

ObjectWindows DLL servers must be compiled with the large memory model. They must be linked with the OLE, ObjectComponents, and ObjectWindows libraries.

The integrated development environment (IDE) chooses the right build options for you when you select *Dynamic Library* for Target Type and request OWL and OCF support from the list of standard libraries. You may choose to link with the static or dynamic versions of the standard libraries.

To build an ObjectWindows DLL server from the command line, create a short makefile that includes the OWLOCFMK.GEN file found in the EXAMPLES subdirectory. Here, for example, is the makefile that builds the sample program Tic Tac Toe:

```
MODELS=ld
SYSTEM = WIN16

DLLRES = ttt
OBJDLL = ttt.obj

!include $(BCEXAMPLEDIR)\owlocfmk.gen
```

DLLRES holds the base name of your resource file and the final DLL name. OBJDLL holds the names of the object files from which to build the sample. Finally, your makefile should include the OWLOCFMK.GEN file.

Name your file MAKEFILE and type this at the command-line prompt:

```
make MODEL=l
```

Make, using instructions in OWLOCFMK.GEN, builds a new makefile tailored to your project. The command also runs the new makefile to build the program. If you change the command to define MODEL as *d*, the above command will create a new makefile and then build your DLL using the DLL version of the libraries instead of the large model static libraries.

For more information about how to use OWLOCFMK.GEN, read the instructions at the beginning of MAKEFILE.GEN, found in the examples directory.

## Debugging a DLL server

The same general techniques used to debug DLLs apply to DLL servers. The steps that follow describe one approach, using Turbo Debugger for Windows to set breakpoints in a DLL server.

1 Build and register the DLL server.
   - Build your server with debugging information.
   - Register the server using the REGISTER.EXE utility (see page 292).
   - Verify that the registration was successful by running RegEdit and looking for your servers file types. (RegEdit is a registration editor included with Windows.)
2 Launch Turbo Debugger for Windows and load a container.

- Select the File | Open menu option and enter the container's name in the Program Name field and click the OK button.

The debugger loads the container. If the container was built without debugging information, you may receive a warning. You can safely ignore it.

**3** Load the server's debugging information.

- Select View | Module from the debugger's main menu. This activates the dialog titled "Load module source or DLL symbols." (You can also activate the module dialog by pressing *F3*.)
- Enter the full name of DLL server file in the DLL Name field.
- Select the Yes option in the Debug Startup field and click the Add DLL button. The name of your DLL server (followed by *!!*) appears as the selected entry in the DLLs & Programs list.
- Click the Symbol Load button.

If you receive an error message indicating that the DLL is not loaded , press the Escape key to return to the debugger's main menu and proceed to the next step. Otherwise, skip the next step and proceed to step 5.

**Note** If you did not receive the error message that the DLL is not loaded, then your DLL server was already in memory before the container activated it. This happens if another container is currently running with one of your server's objects. More often, however, it indicates that your server crashed or was improperly terminated in an earlier session.

**4** Run the container and insert one of your server's objects.

- Select the Run | Run menu option (or press *F9*) to start the container.
- Choose Insert Object from the container's Edit menu and insert your server's object.

The debugger pops up as soon as OLE loads your DLL server.

**5** Display the DLL source modules and set breakpoints.

- Choose View | Module from the container's main menu to see the names of the source files used to build your server. The file names appear in Source Modules list.
- Select source files by double-clicking the file names.
- Set breakpoints in your server.
- Choose the Run | Run menu option (or press *F9*) to return control to the container application.

**6** If you skipped step 4, insert one of your server's objects into the container now.

The debugger stops at the breakpoints set in your source files and allows you to step through your server, inspect variables, and verify the logic of your code.

# Tools for DLL servers

Before running your DLL server, you must record its registration information in the system registration database. The Register tool does that for you. Another tool, DllRun, gives you the option of running your DLL server at any time as a standalone application, which is sometimes convenient for testing.

## Registering your DLL server

The REGISTER.EXE utility registers an ObjectComponents DLL server. On the command line, pass Register the name of your server followed by the -RegServer switch. Here is the command to register Tic Tac Toe:

```
register ttt.dll -RegServer
```

Even though the Register utility is a Windows application, not a DOS application, you can invoke it from a Windows DOS box. This ability is useful in makefiles. (To invoke other Windows programs from a DOS box command line, use the WinRun utility described in UTILS.TXT.)

Register can also unregister your server. Unregistering removes all entries related to your server from the registration database. It's good practice to unregister one version before you register the next. To unregister, use the -UnregServer switch. This command unregisters Tic Tac Toe:

```
register ttt.dll -UnregServer
```

## Running your DLL server

The DLLRUN.EXE utility lets you load and run an ObjectComponents DLL server as though it were a standalone executable program. The ability to run in executable mode is useful for debugging. It also lets you give customers the choice of running your server either way without having to distribute two versions of the same application.

On the command line, pass DllRun the *progid* of the server. This is the value assigned to the *progid* key in the server's registration table. This command runs the Tic Tac Toe server:

```
dllrun TicTacToeDll
```

DllRun launches the DLL server in the executable running mode. The running mode of an ObjectComponents application is represented by a set of bit flags that you can test by calling *TOcModule::IsOptionSet*. (Remember that the application object of a linking and embedding program derives from both *TApplication* and *TOcModule*.)

The running mode bit flags are defined in the *TOcAppMode* **enum**. *amEmbedded* is set when the server is invoked by OLE, not by the user. *amExeModule* is set in an application that was built as an EXE. *amExeMode* is set in an application that is running as a standalone executable, even if it was built as a DLL.

This code tests the flags to determine the server's running mode.

```
void
TMyApp::TestMode()
{
```

```
  if (IsOptionSet(amExeMode))             // is server running as an EXE?
    if (!IsOptionsSet(amExeModule)) {     // if so, was it built as an EXE?
      // the server is a DLL running in EXE mode
    } else {
      // the server was built as an EXE
  } else {
    // the server is a DLL running in a client's process
}
```

Chapter

21

# Automating an application

Automating a program means exposing its functions to other programs. Once a program is automated, other programs can control it by issuing commands through OLE. ObjectComponents is your interface to OLE automation. Through ObjectComponents you can expose any C++ class to OLE, and you won't have to restructure your existing classes to do it.

The process of automating an application is the same whether the application uses ObjectWindows or not. The steps described in this chapter work for any C++ application.

This chapter explains how to do the following things:

- Automate an application
- Expose enumerated values
- Install a hook to validate arguments in a command
- Combine multiple C++ objects into a single automated OLE object
- Invalidate an automation object when it is destroyed
- Use localized strings to match the controller's language setting
- Expose a collection of objects
- Create and register a type library

Automation servers can be built as DLLs using the same methods for making an in-process linking and embedding server. See "Making a DLL server" on page 374.

## Steps to automating an application

These are the coding steps that make a program automatable. The sections that follow explain each step in more detail.

1 Register the application.
   - Create a registration table using registration macros.
   - Create a registrar object and call its *Run* method.

2 In a declaration table, declare the methods and properties you want to expose.
   - Optionally, provide hook functions to record, filter, or undo commands and to validate command arguments.
3 In a definition table, assign public names for the methods and properties you want to expose.
4 Build the application.
   - Include automation header files in your source code.
   - Link to the OLE and ObjectComponents libraries.

Two other optional steps are sometimes useful:

- Provide a resource table of translated strings so that OLE can accommodate whatever language setting the user has selected.
- Notify OLE when your object is destroyed. This is only a safety feature and is often superfluous.

ObjectComponents lets OLE reach members of your classes through standard C++ mechanisms. At run time, other programs can send commands for your program to OLE.

A program that exposes itself to receive automation commands is called an *automation server*. A program that sends commands for others to execute is called an *automation controller*. (Chapter 21 explains how to make a controller.)

Many of the code examples in this chapter are based on the AutoCalc example program installed in the EXAMPLES/OCF/AUTOCALC directory. AutoCalc draws a calculator on the screen and lets the user click buttons to perform calculations. AutoCalc also automates its classes so that a controller application can send commands to do the same things a user does.

# 1. Registering an automation server

Registering an application means giving OLE information about what the application can do. First you record the information in a registration table, and then you pass the table to the constructor of a registrar object.

## Creating a registration table

An automation server must set four pieces of information in the application's registration table: its program ID, its class ID, a description of itself, and command-line arguments for invoking the automation server.

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,      "{877B6200-7627-101B-B87C-0000C057CE4E}")
  REGDATA(progid,     "Calculator.Application")
  REGDATA(appname,    "AutoCalc"
  REGDATA(description,"Automated Calculator 1.2 Application")
  REGDATA(cmdline,    "/Automation")
END_REGISTRATION
```

The registration macros create a structure that this example names *AppReg*. The *clsid* must be specially generated to ensure uniqueness. To learn how, refer to the entry for the *clsid* registration key in the *ObjectWindows Reference Guide*. The *progid* for an automation server conventionally has two parts, one naming the program ("Calculator") and one naming the type of object ("Application"). A period (".") is the only permissible delimiter character in a *progid*.

A server that creates several different kinds of automatable objects must give each a different *progid*, such as *AppName.MySecondObject* and *AppName.MyThirdObject*. You need not, however, supply a different *clsid* for each kind of object, only for the first one. ObjectComponents increments the first *clsid* for each subsequent object.

The *progid* is visible to users when they create your object in their automation scripts. They also see the *description* string when OLE browses the system for available automation objects.

For the final registration key, *cmdline*, an automation server should normally include the -Automation switch. When an automation controller asks to create your object, OLE invokes your application and places the *cmdline* value on the command line.

Although OLE conventions call for this switch, OLE itself pays no attention to it. ObjectComponents uses it, though, to determine whether to invoke new instances of your program for each new OLE client or whether a single instance should service all clients. Normally you don't want several different clients sending commands to the exact same object. The commands they send might interfere with each other. ObjectComponents responds to the -Automation switch by making the application support only one client per instance. (The -Automation switch overrides *ocrMultipleUse* if you register that in your *usage* key.) If it makes sense for your automation server to support simultaneous clients with a single instance, then register it as *ocrMultipleUse* and omit -Automation from the *cmdline* string.

The -Automation switch directs the creation of an OLE factory, the facility that ObjectComponents registers at run time for OLE to call when it wants to create whatever the server produces. When the switch is present, ObjectComponents registers the application class so that OLE can create an instance of the application. ObjectComponents does not register the document factory. To make the application single-use, ObjectComponents removes the factory after it creates the first instance of the application.

The registration table in the example holds information about the application, so it is called an *application registration table*. Using the same macros, an application can also create *document registration structures* to describe the kinds of documents (or objects) that the server produces. An automation server creates automated documents if it wants controllers to embed the automated object before issuing commands. For more information about application and document registration structures, see "Building registration tables" on page 344.

Table 21.1 briefly describes all the registration keys that might be used by an automation server. It shows which are optional and which required, as well as which belong in the

application registration table (usually named *AppReg*) and which in the document registration table (usually named *DocReg*).

**Table 21.1** Keys an automation server registers

| Key | In AppReg? | In DocReg? | Description |
|---|---|---|---|
| appname | Yes | Optional | Short name for the application |
| clsid | Yes | Optional | Globally unique identifier (GUID); generated automatically for the *DocReg* structure |
| description | Yes | Yes | Descriptive string (up to 40 characters) |
| progid | Yes | Yes | Name of program or object type (unique string) |
| extension | No | Optional | Document file extension associated with server |
| docfilter | No | Yes if not *dtHidden* | Wildcard file filter for File Open dialog box |
| docflags | No | Yes | Options for running the File Open dialog box |
| typehelp | No | Optional | Name of an .HLP file documenting supported commands |
| helpdir | No | Optional | Path to an .HLP file documenting supported commands (defaults to current module path) |
| debugger | Optional | Optional | Command line for running debugger. |
| debugprogid | Optional | Optional | Name of debugging version (unique string) |
| debugdesc | Optional | Optional | Description of debugging version |
| cmdline | Yes | No | Arguments to place on server's command line |
| path | Optional | No | Path to server file (defaults to current module path) |
| permid | Optional | Optional | Name string without version information |
| permname | Optional | Optional | Descriptive string without version information |
| usage | Optional | Optional | Support for concurrent clients |
| language | Optional | No | Language for registered strings (defaults to system's user language setting) |
| version | Optional | No | Major and minor version numbers (defaults to "1.0") |

The table assumes that the server's documents support automation. For non-automated document types, the server needs to register only *docflags* and *docfilter*.

If your server is also an OLE container or a linking and embedding server, then you should also consult the container table (page 308) or the server table (page 346). Register all the keys that are required in any of the tables that apply to your application.

For more information about individual registration keys and the values they hold, see the *ObjectWindows Reference Guide*.

An automation server that supports system language settings should localize the *description* string it registers. (The *progid* must never be localized.) For more on localization, skip ahead to "Localizing symbol names" on page 397.

The complete *AppReg* structure is later passed to the program's *TRegistrar* object and written to the registry.

### Creating a registrar object

An automation server needs a registrar object, just as linking and embedding applications do. Applications that support only automation, however, without linking and embedding, should create *TRegistrar* instead of *TOcRegistrar*. *TRegistrar* is the base class for *TOcRegistrar*. *TOcRegistrar* extends *TRegistrar* by connecting the application to the BOCOLE support library interfaces that support linking and embedding.

First, declare a static pointer to hold the *TRegistrar**. Use the *TPointer<>* template to ensure that the registrar object is properly deleted when the program ends.

```
TPointer<TRegistrar> Registrar;    // initialized at WinMain or LibMain
```

In the main procedure (for AutoCalc, this is *WinMain*), you should create the registrar object and call its *Run* method.

```
try {
  ::Registrar=new TRegistrar(AppReg;TOcAutoFactory<TCalc>,string(cmdLine),hInst);
  if (!::Registrar->IsOptionSet(amAnyRegOption))
    ::Registrar->Run();
  ::Registrar = 0;                    // deletes registrar by replacing pointer
  return 0;
}
catch (TXBase& x) {
  ::MessageBox(0, x.why().c_str(), "OLE Exception", MB_OK);
}
```

The first parameter of the *TRegistrar* constructor is the application registration structure, conventionally named *AppReg*. The second parameter is a factory callback function. The example uses a factory template to create the callback. For an automation server that doesn't use ObjectWindows, the appropriate template is *TOcAutoFactory*. (For more about factory callbacks and templates, see the *ObjectWindows Reference Guide*.)

The call to *IsOptionSet* determines whether the application was passed a command-line switch asking the application to register itself in the system registration database and then quit. If not, the application calls *Run*. The registrar then calls the factory callback, where the message loop resides. When *Run* returns, the application has ended.

For more information about command-line switches, see Table 20.2.

## 2. Declaring automatable methods and properties

Automating a class requires building two tables, one in the class declaration and one in the class implementation. The first table is called the *automation declaration*, and it declares which members of the class a controller can reach. The second table is called the *automation definition*, and it defines public names that a controller uses to reach each exposed class member. This section tells how to build an automation declaration.

The automation declaration belongs inside the declaration of an automated class. It begins with the macro DECLARE_AUTOCLASS and includes one entry for each class member that you choose to expose. The macros add nested classes that ObjectComponents instantiates to process commands received from OLE. They do not alter the structure or size of the original class.

This sample automation declaration exposes functions and data members of a C++ class that mimics a calculator:

```
DECLARE_AUTOCLASS(TCalc)
  AUTODATA   (Accum,   Accum,     long,  )
  AUTODATA   (Opnd,    Opnd,      long,  )
  AUTODATA   (Op,      Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_CLEAR))
  AUTOFUNC0  (Eval,    Eval,      TBool, )
  AUTOFUNC0V(Clear,    Clear,            )
  AUTOFUNC0V(Display, Display,           )
  AUTOFUNC0V(Quit,     Quit,             )
  AUTOFUNC1  (Button,  Button,    TBool, TAutoString,)
  AUTOFUNC0  (Window,  GetWindow, TAutoObject<TCalcWindow>,  )
  AUTOFUNC1  (LookAt,  LookAtWindow, long, TAutoObject<const TCalcWindow>,)
  AUTODATARO(MyArray, Elem,       TAutoObjectByVal<TMyArray>,)
```

The automated class is called *TCalc*. Each AUTOFUNC or AUTODATA macro exposes one member of *TCalc*. Some of the *TCalc* member functions are *Eval*, *Clear*, *Display*, and *Quit*. Its data members include *Accum*, *Opnd*, *Op*, and *Elem*. *TCalc* also has other members that it chooses not to automate and so excludes from the declaration table.

No termination macro is needed for an automation declaration. The END_AUTOCLASS macro that closes an automation definition is not used here. Also, each line of the declaration ends with a closing parenthesis, not with punctuation.

**Note** The automation declaration should appear at the end of a class declaration because the macros can modify the access specifier. If you put the declaration anywhere other than the end, be sure to follow it immediately with an access specifier (**public**, **protected**, or **private**).

## Writing declaration macros

Each of the macros within an automation declaration describes a single method or property that other programs can manipulate. The different macros expose different kinds of class members. AUTOFUNC1, for example, exposes a member function that takes one parameter. AUTOFUNC2V exposes a function that takes two parameters and returns nothing (**void**). AUTOPROP exposes a property through Set and Get functions that insert or retrieve a single value. AUTODATA exposes a data member that the controller can read and modify directly. For a complete list of the macros, refer to the table of Automation Declaration Macros in Chapter 5 of the *ObjectWindows Reference Guide*.

The general form of the automation macros is this:

```
MACRONAME( InternalName, FunctionName, ReturnType, ArgumentType, Options )
```

Some of the macros don't use all five parameters. AUTOFUNC1V, for example, doesn't have a *ReturnType* because the function has a **void** return. AUTOFUNC0 doesn't have any arguments, while AUTOFUNC2 has two different arguments. But whatever parameters are relevant appear in the order shown.

*InternalName* is an identifier you assign to each automatable property or function. It is used internally by ObjectComponents for keeping track of the members. The only other place you ever use the internal name is in the corresponding entry of the class's

automation definition table. The internal name is a unique identifier for the member. (The names used in source code are not necessarily unique. They can be overloaded, for example.)

*FunctionName* is the name you use in your source code to refer to the same property or function. *FunctionName* can be any expression that evaluates to a function call. The expression must, however, be defined within the scope of the automated object. ObjectComponents attempts to reach the function through the **this** pointer.

The internal and function names should be the same unless the function name is overloaded or uses indirection. For example, suppose a class contains a data member that points to another object:

```
TObject* MyObject;
```

To expose a function call like *MyObject->MyFunction*, you should supply an internal name that does not use indirection. In this case, a good choice would be *MyFunction*.

```
AUTOFUNC0V( MyFunction, MyObject->MyFunction, )
```

If a function is overloaded, use the same function name for all versions but give each a different internal name. ObjectComponents can distinguish the overloaded functions by the return types and argument types in the parameters that follow.

The *ReturnType* and *ArgumentType* parameters can be any fundamental C type, such as **int** or **char**, or a pointer to any fundamental type. Some pointers, however, require special handling. If the data type is a string (type **char***), declare it to be a *TAutoString* instead. If the data type is a pointer or a reference to a C++ object, then declare it using the *TAutoObject<>* wrapper. The type substitutions help ObjectComponents convert between C++ data types and the VARIANT union type that OLE uses. Pointers and object references are hardest to convert because they refer to data that is not in the variable itself. The *TAutoString* and *TAutoObject* classes provide type information for the conversion so that ObjectComponents can pass the right information between server and controller applications.

The *TCalc* example shows how to use *TAutoObject*. One of the functions *TCalc* exposes is *GetFunction*, which returns a reference to a *TCalcWindow* object.

```
AUTOFUNC0 (Window, GetWindow, TAutoObject<TCalcWindow>, )
```

When it declares *TCalcWindow* as the return type, it makes use of the *TAutoObject* template to create a smart, self-describing pointer to a *TCalcWindow* object.

## Providing optional hooks for validation and filtering

The final parameter of every automation macro names a hook function to be called whenever OLE calls the exposed class member. A *hook* is code that executes every time anyone uses a particular class member. ObjectComponents supports hooks to record commands, undo commands, validate command arguments, and override a command's implementation. Hooks are always optional.

To install a hook, use one of these macros as the last parameter to any automation declaration:

- AUTOINVOKE
- AUTORECORD
- AUTOUNDO
- AUTONOHOOK
- AUTOREPORT
- AUTOVALIDATE

Each macro receives a single parameter containing code to execute. The form of the required macro varies with its function.

To validate arguments, for example, the code should be a Boolean expression. The *Op* data member of *TCalc* holds an integer that identifies an operation to perform, such as addition or subtraction. The automation declaration installs a hook to be sure that *Op* is not assigned a value outside the legal range of operator identifiers.

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_CLEAR))
```

AUTOVALIDATE introduces the expression to execute for validation. Within the validation expression, use the name *Val* to represent the value received from the controller. When used to validate function arguments, AUTOVALIDATE uses the names *Arg1*, *Arg2*, *Arg3*, and so on.

Whenever any automation controller attempts to set a value in the *Op* data member, ObjectComponents verifies that the new value falls within the range OP_NONE to OP_CLEAR. If passed an illegal value, ObjectComponents cancels the command and sends OLE an error result.

The expression passed to AUTOVALIDATE can include function calls.

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && NotTooBig(Val))
```

Now ObjectComponents calls *NotTooBig* whenever a controller attempts to modify *Op*.

```
bool NotTooBig(int Val) {
  return (Val <= OP_CLEAR)
}
```

## 3. Defining external methods and properties

Besides declaring which of its members are automatable, an automated class must also create a second table of macros to assign public symbols for referring to the exposed methods and properties. The public symbols are what other applications see. They become the controller's interface to an automated OLE object.

Behind the scenes, ObjectComponents links the public names to the C++ object or objects that you create to implement the OLE object. The automation declaration table identifies which class members to expose, and the automation definition table assigns them names.

The automation definition belongs with the class implementation. It begins with the DEFINE_AUTOCLASS macro and ends with END_AUTOCLASS. Here's the automation definition for *TCalc*:

```
DEFINE_AUTOCLASS(TCalc)
  EXPOSE_PROPRW(Opnd,      TAutoLong,  "Operand",      "@Operand_", HC_TCALC_OPERAND)
```

```
  EXPOSE_PROPRW_ID(0,Accum,TAutoLong,  "!Accumulator", "@Accumulator_",
        HC_TCALC_ACCUMULATOR)
  EXPOSE_PROPRW(Op,         CalcOps,    "Op",           "@Op_", HC_TCALC_OPERATOR)
  EXPOSE_METHOD(Eval,       TAutoBool,  "!Evaluate",    "@Evaluate_", HC_TCALC_EVALUATE)
  EXPOSE_METHOD(Clear,      TAutoVoid,  "!Clear",       "@Clear_", HC_TCALC_CLEAR)
  EXPOSE_METHOD(Display,    TAutoVoid,  "!Display",     "@Display_", HC_TCALC_DISPLAY)
  EXPOSE_METHOD(Quit,       TAutoVoid,  "!Quit",        "@Quit_", HC_TCALC_QUIT)
  EXPOSE_METHOD(Button,     TAutoBool,  "!Button",      "@Button_", HC_TCALC_BUTTON)
   REQUIRED_ARG(             TAutoString,"!Key")
  EXPOSE_PROPRO(Window,     TCalcWindow,"!Window",      "@Window_", HC_TCALC_WINDOW)
  EXPOSE_METHOD(LookAt,     TAutoLong,  "!LookAtWindow","@LookAtWindow_",
        HC_TCALC_LOOKATWINDOW)
   REQUIRED_ARG(             TCalcWindow,"!Window")
  EXPOSE_PROPRO(MyArray,    TMyArray,   "!Array",       "@Array_", HC_TCALC_ARRAY)
  EXPOSE_APPLICATION(       TCalc,      "!Application", "@Application_",
        HC_TCALC_APPLICATION)
END_AUTOCLASS(TCalc, tfNormal, "TCalc", "@TCalc", HC_TCALC)
```

The EXPOSE_*xxxx* macros assign names to methods and properties. EXPOSE_PROPRW defines a property that controllers can both read and write. EXPOSE_PROPRO limits a controller's access so it can only read the property value. REQUIRED_ARG assigns a name to a function argument.

For example, a controller invokes the *LookAt* function by using the name *LookAtWindow*, and it calls the function's one parameter *Window*. The DEFINE_AUTOCLASS and END_AUTOCLASS macros assign "TCalc" as the public name for objects of type *TCalc*.

Most of the strings in this automation definition begin with a symbol, either ! or @. These symbols indicate that the AutoCalc application has in its resources translations for each public symbol. Each command from an automation controller comes with a locale ID indicating the language the controller is using. If the controller was written in German, for example, it can pass the string "Auswerten" instead of "Evaluate," and ObjectComponents correctly invokes the *Eval* function. For more about using international strings, see "Localizing symbol names" later in this chapter.

Every item listed in the automation definition must already appear in the automation declaration. For example, every function name you define with EXPOSE_METHOD must have a corresponding AUTOFUNC declaration. Every EXPOSE_PROP must have a corresponding AUTOPROP, AUTOFUNC, AUTOFLAG, or AUTODATA, depending on how you implement the property.

## Writing definition macros

The macros for exposing methods and properties have five parameters: the internal name, the type of value returned, the external name, and a documentation string. The optional fifth parameter allows you to associate a Help context ID with each member.

```
MACRONAME(InternalName, ReturnType, ExternalName, DocString, HelpContext )
```

- *InternalName* is the identifier string you assigned to the member in the automation declaration.

- *ReturnType* tells what automation data type the method returns or the property holds. Automation data types are listed in the last column of Table 21.2 later in this chapter.
- *ExternalName* is what automation controllers see. A user sending commands from a controller refers to all properties and methods by their external names.
- *DocString* should explain to a user what the exposed property or method does. OLE displays this string if the user asks for help with a particular automation command. If you omit the document string, set the parameter to 0.
- *HelpContext*, the fifth parameter, is optional. It is a number that identifies a particular section of a Windows Help file (.HLP). You can create a Help file that describes the syntax and usage of all the members you expose. If you supply the context IDs for each member in the class's automation definition, then an automation controller can ask OLE to display the help screens for the user. A user writing an automation script, for example, can browse at run time for the list of members your application exposes, ask to see their document strings, and even ask to see a Help screen about each one.

  If you provide a Help file for automation, you should be sure to register its name with the *typehelp* key, described in Table 21.1.

For a complete list of the different EXPOSE macros, see the table of Automation Definition Macros in Chapter 5 of the *ObjectWindows Reference Guide*.

When exposing a method that takes arguments, you also need to add to the definition a macro describing each argument. Here is the prototype for a function that takes three arguments, along with the macros needed to define the method for automation:

```
// member function declaration
long TCalculator::AddNumbers(short Num1, short Num2 = 0, short Num3 = 0);

// later, this appears after DEFINE_AUTOCLASS(TCalculator)
EXPOSE_METHOD(AddNumbers, TAutoLong, "AddNumbers", "Sum up to 3 numbers", HC_ADDNUMBERS)
  REQUIRED_ARG(TAutoShort, "Num1")
  OPTIONAL_ARG(TAutoShort, "Num2", "0")
  OPTIONAL_ARG(TAutoShort, "Num3", "0")
```

The first argument, *Num1*, is required. The others are optional. All three are **short** integers. When describing optional arguments, you need to supply a default value. In the example, *0* is the default value for the two optional arguments.

OLE conventions suggest that each automation object should have a property representing the application it belongs to. You can add this property to any automation definition with the EXPOSE_APPLICATION macro.

```
EXPOSE_APPLICATION(TMyClass, "Application", "My Application",)
```

The class passed to EXPOSE_APPLICATION must be the same class passed to the factory template, as shown in "Creating a registrar object" on page 385.

## Data type specifiers in an automation definition

Most of the macros in an automation definition ask for a data type—the type of a function's return value, of each function argument, or of a data member. The possible

values for data types within an automation definition are not fundamental C types. They can be any of the following:

- An enumeration value previously defined for automation. (This technique is explained in the next section.)
- The name of an automated class (such as *TCalc*).
- Any of the predefined classes that ObjectComponents provides to represent intrinsic C types. See Table 21.2.

The reason for exposing predefined classes rather than intrinsic C types is to make type information available when browsing from the controller. For exposed classes, ObjectComponents can extract type information using RTTI. The automation data types in the following table are defined as structures that contain no data; they simply retrieve a static value indicating a data type. The identifier values are the same identifiers that OLE uses to distinguish the data types it supports. All the automation data types derive from a base called *TAutoVal*, so they are polymorphic. In effect, ObjectComponents can ask any value passed through automation to describe its own data type.

Table 21.2 lists all the automation data types and shows where to use them. Start with the left column and find a type that your automated class uses in its arguments or its return values. The other columns tell what data type to specify in the corresponding entries of the automation declaration and definition.

**Table 21.2** Automation data types

| C++ type (in class) | Transfer type (in DECLARE_AUTOCLASS) | Exposed type (in DEFINE_AUTOCLASS) |
|---|---|---|
| short | short | TAutoShort |
| unsigned short | short or unsigned | TAutoShort or TAutoLong |
| long | long | TAutoLong |
| unsigned long | unsigned long | TAutoLong (treated as signed long) |
| int | int | TAutoInt |
| unsigned int | int or long | TAutoInt or TAutoLong |
| float | float | TAutoFloat |
| double | double | TAutoDouble |
| bool (or int) | TBool | TAutoBool |
| TAutoDate | TAutoDate | TAutoDate |
| TAutoCurrency | TAutoCurrency | TAutoCurrency |
| char* | TAutoString | TAutoString |
| const char* | TAutoString | TAutoString |
| char far* | TAutoString | TAutoString |
| const char far* | TAutoString | TAutoString |
| string | string | TAutoString |
| enum | short or int | TAutoShort, TAutoInt, or user-defined AUTOENUM |
| T* | TAutoObject<T> | T (class T must be automated) |
| T& | TAutoObject<T> | T (class T must be automated) |
| const T* | TAutoObject<const T> | T (class T must be automated) |

**Table 21.2** Automation data types (continued)

| C++ type (in class) | Transfer type (in DECLARE_AUTOCLASS) | Exposed type (in DEFINE_AUTOCLASS) |
|---|---|---|
| const T& | TAutoObject<const T> | T (class T must be automated) |
| T* (returned) | TAutoObjectDelete<T> | (C++ object deleted if no refs) |
| T& (returned) | TAutoObjectDelete<T> | (C++ object deleted if no refs) |
| T (returned) | TAutoObjectByVal<T> | T (T copied, deleted when refs==0) |
| void (no return) | (use AUTOFUNCxV macros) | TAutoVoid |
| short far* | short far* | TAutoShortRef |
| long far* | long far* | TAutoLongRef |
| float far* | float far* | TAutoFloatRef |
| double far* | double far* | TAutoDoubleRef |
| TAutoDate far * | TAutoDate far* | TAutoDateRef |
| TAutoCurrency far* | TAutoCurrency far* | TAutoCurrencyRef |

## Exposing data for enumeration

An automation server might also need to expose enumerated values. Use OLE enumerations when you want to expose a set of internal data values and refer to them with localizable strings. For example, AutoCalc defines the enumerated type operators to represent different actions the calculator can perform with numbers.

```
enum operators {
  OP_NONE = 0,
  OP_PLUS,
  OP_MINUS,
  OP_MULT,
  OP_DIV,
  OP_EQUALS,
  OP_CLEAR,
};
```

As the calculator receives input, it stores the pending mathematical operation in a private data member called *Op*.

```
short Op;
```

Operations are identified by different OP_*xxxx* constants. The *Eval* method performs the pending operation using the number just entered and the total in the calculator's accumulator. AutoCalc exposes the *Op* data member to automation so that a controller can enter operators directly. Here's the automation declaration:

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_CLEAR))
```

The automation declaration shows that the *Op* data member holds a short value, but the symbols OP_PLUS and OP_MINUS are defined only within the server program. The controller can't use them when it passes commands. Ideally the controller should be able to use more readable strings such as "Add" and "Subtract" in scripts.

The place for declaring public symbols is the automation definition. Use the DEFINE_AUTOENUM macro to begin a table defining symbols for the enumerated values.

```
DEFINE_AUTOENUM(CalcOps, TAutoShort)
  AUTOENUM("Add",      OP_PLUS)
  AUTOENUM("Subtract", OP_MINUS)
  AUTOENUM("Multiply", OP_MULT)
  AUTOENUM("Divide",   OP_DIV)
  AUTOENUM("Equals",   OP_EQUALS)
  AUTOENUM("Clear",    OP_CLEAR)
END_AUTOENUM(CalcOps, TAutoShort)
```

The AUTOENUM macro takes two parameters: an enumeration string and a constant value. The enumeration string (which can be localized) is the external name exposed through OLE for use by controllers.

The macros that begin and end the enumeration table assign the name *CalcOps* to this enumerated type. They also associate the automated data type *TAutoShort* with this enumeration because the enumerated values are all **short ints**.

Table 21.3 lists the C++ types that can be enumerated and the corresponding automation types for exposing them.

**Table 21.3** Enumerable C++ types and the automation types for exposing them

| C++ type | Enumeration type (for automation definitions) |
|---|---|
| bool | TAutoBool |
| double | TAutoDouble |
| float | TAutoFloat |
| int | TAutoInt |
| long | TAutoLong |
| short | TAutoShort |
| const char* | TAutoString |

Creating a table of enumerated values results in a new data type that you can use to describe arguments and return values in an automation definition. Now that ObjectComponents understands the *CalcOps* enumerated type, you can use the type to define the *Op* property.

```
EXPOSE_PROPRW(Op, CalcOps, "Op", "@Op_", HC_TCALC_OPERATOR)
```

This line says that *Op* is a read-write property holding a value of type *CalcOps*. When the controller tries to place "Multiply" or "Divide" in the *Ops* property, ObjectComponents correctly translates the string into the value defined as OP_MULT or OP_DIV.

## 4. Building the server

To build an automation server, you need to include the right headers and link to the right libraries.

### Including header files

An automated program needs to include the following headers:

```
#include <ocf/automacr.h>   // definition and declaration macros
#include <ocf/ocreg.h>      // TRegistrar class
```

The list is short because an automation server does not need many of the ObjectComponents classes used for linking and embedding.

### Compiling and linking

Automation servers and controllers must be compiled with the medium or large memory model. (They run faster in medium model.) They must be linked with the OLE and ObjectComponents libraries.

The IDE chooses the right build options for you when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OCFMAKE.GEN file found in the EXAMPLES subdirectory.

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj myprogram.obj
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJRES hold the name of the file to build and the names of the object files to build it from. The last line includes the OCFMAKE.GEN file. Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=l
```

MAKE, using instructions in OCFMAKE.GEN, will build a new makefile tailored to your project. The new makefile is called WIN16L*xx*.MAK.

For more information about OCFMAKE.GEN and the libraries needed for an ObjectComponents program, see "Building an ObjectComponents application" on page 286.

**Note** The first time the server runs, the registrar object records its information in the registration database. Be sure to run the server once before trying to use it with a controller.

## Enhancing automation server functions

The preceding sections have explained what additional code different kinds of applications must provide in order to become automation servers. The remaining sections explain ways to enhance a server's capabilities. Enhancements include:

- Combining several C++ classes into one automation object
- Invalidating an automation object when it goes away
- Localizing symbol names and registration entries
- Exposing collections of objects
- Making type libraries

## Combining multiple C++ objects into a single OLE automation object

The complete set of member functions and properties that belong to a single automatable OLE object can in fact be implemented by a combination of C++ objects. An automatable calendar, for example, might begin with a *TCalendar* class. But the automatable OLE calendar object might need to expose some methods and properties that don't happen to belong to the C++ *TCalendar* object. The background color, for example, might be inherited from *TCalendar*'s base class, and some of the input functions might belong to separate control windows in the calendar's client area. In that case, the automation declaration for *TCalendar* should delegate some tasks to other C++ classes. To combine several C++ objects together into a single OLE object, add macros to the automation definition table.

```
// these lines belong in the definition block that begins DEFINE_AUTOCLASS(TCalendar)
EXPOSE_INHERIT(TCalendarWindow, "CalendarWindow")
EXPOSE_DELEGATE(TWeekForwardButton, "WeekForward", GetWeekForwardButton(this))
```

Any exposed classes must also be automated. In other words, *TCalendarWindow* and *TWeekForwardButton* must also have their own AUTOCLASS tables. By exposing both of these classes in the *TCalendar* automation definition, you combine all the exposed members from all three classes into a single symbol table. When OLE sends an automation command to the calendar, ObjectComponents searches for the matching class member in *TCalendar*, then in *TCalendarWindow*, and finally in *TWeekForwardButton*.

The EXPOSE_DELEGATE macro takes as its third parameter a conversion function. In order to reach members in the delegation class, ObjectComponents needs a pointer to an object of that class. The conversion function has one parameter for receiving a this pointer to the object where the definition table appears. The function must return a pointer to the delegation object. Also, it must be a global function. For example, if *TCalendar* has a data member that points to the Week Forward button, this might be the conversion function.

```
TWeekForwardButton *GetWeekForwardButton ( TCalendar* this ) {
  return( this->m_ForwardButton );
}
```

You don't need to provide a conversion function when exposing an inherited function or property because in that case ObjectComponents can create its own templatized conversion function to reach the base class.

**Note** Another way to coordinate the actions of several automated objects within a single application is to give one object access functions that return the other objects. For example, the sample program AutoCalc automates five different classes, but no class delegates to any other. When a controller asks for an object from the AutoCalc server, it receives only the automated *TCalc* object. *TCalc*, however, has a property called *Window* that holds a *TCalcWindow* object. *TCalcWindow*, in turn, has a property that holds the collection of buttons. The collection object returns individual button objects. Without properties or functions that return the other objects, the controller would never be able to reach them. Be sure to add access functions if necessary.

To find out how a controller uses the access functions, see "Sending commands to the collection" on page 415.

## Telling OLE when the object goes away

If there is a chance that your program might delete its automated object while still connected to a controller, then you need to tell OLE when the object is destroyed. This precaution matters only if the logic of your program might cause the object to be destroyed through nonautomated means while an OLE session is still in progress. If OLE attempts to use an automation object whose underlying C++ object has been destroyed, it attempts to use an invalid pointer. A single function call prevents the error by sending OLE an obituary to announce that the object no longer exists.

```
// place this line in the destructor of your automated class
::GetAppDescriptor()->InvalidateObject(this);
```

*GetAppDescriptor* is a global function returning a pointer the application's *TAppDescriptor* object. *InvalidateObject* is a *TAppDescriptor* method. It tells OLE the object that was passed to the descriptor's constructor is now invalid.

Although the object's destructor is a good place to call *InvalidateObject*, you can call it anywhere. If you do not own the class you are automating, it might not be possible to modify the destructor. This works, too:

```
TMyAutoClass MyAutomatedObject;
:
::GetAppDescriptor()->InvalidateObject(MyAutomatedObject);
delete MyAutomatedObject;
```

The object pointer you pass to *InvalidateObject* must always represent the most derived form of the object. In other words, if the pointer is polymorphic, it must point to the class as it was created and not to any of its base classes. Calling *InvalidateObject* from the object's own destructor is safe because in that case **this** always points to the most derived class. If you call *InvalidateObject* from somewhere else, you might need the global function *MostDerived* to ensure that you are invalidating the correct object.

```
appDesc->InvalidateObject(::MostDerived(MyPolymorphObject, typeid(MyPolymorphObject)));
```

In the example, *MyPolymorphObject* is a pointer to a polymorphic object, so it might point to a base class or to an object of any type derived from the base. *MostDerived* converts the pointer, making it point to an object of the type furthest down the hierarchy, the one furthest descended from the base.

Besides calling *InvalidateObject*, there are two other ways to be sure OLE knows when the object is destroyed. One way is to derive the object's class from *TAutoBase*. The only code in *TAutoBase* is a virtual destructor that calls *InvalidateObject* for you. This example declares a class called *TMyAutoClass*. OLE always knows when any object of type *TMyAutoClass* is destroyed.

```
class TMyAutoClass: public TAutoBase { /* declarations */ };
```

The other way is to put the AUTODETACH macro in the class's automation declaration table. This works without having to change the class derivation, but it does add one byte to the size of the class.

## Localizing symbol names

The symbols that appear in an automation definition become visible to other OLE programs. Users writing scripts can see and use the symbols. The symbols become part of the program's user interface. Programs intended to reach international audiences need to translate the strings for different markets. For example, a property named "Color" in English should be called "Couleur" in a French script, "Farbe" in a German script, and "Colour" in a British one.

OLE does its best to help you out by passing a number that indicates the user's language setting. This number is called a locale ID, or LCID. LCIDs are defined by OLE and the Win32 API. They consist of two numbers, one identifying a language and one identifying a subdialect within the language. When OLE passes an automation call into an automated application, it also passes an LCID. The automation controller might determine the LCID from the system settings at run time, or the person using the controller might choose a locale.

An automated program is expected to examine the LCID and respond with appropriately translated strings. ObjectComponents eases the burden by letting you build a resource table to supply localized versions of any strings you use. When handling automation calls, ObjectComponents automatically searches the table to find strings that match whatever language the controller requests.

ObjectComponents searches first for a string with the correct language and dialect IDs. Failing that, ObjectComponents searches for a match on primary language only, ignoring dialect. If still no match is found, ObjectComponents simply uses the original, untranslated string.

Localizing your symbols takes two steps.

1 Supply translations for your strings in your program's resources.
2 In your source code, mark the strings that have translations.

### Putting translations in the resource script

To build a table of translations in your resource (.RC) file, use the XLAT resource type.

```
#include "owl/locale.rh"

Left    XLAT FRENCH "Gauche" GERMAN "Links" SPANISH "Izquierda" XEND
Right   XLAT FRENCH "Droit"  GERMAN DUTCH "Rechts"              XEND
Center  XLAT ENGLISH_UK FRENCH GERMAN "Centre" SPANISH "Centro" XEND
Help    XLAT FRENCH "Aide" GERMAN "Hilfe" SPANISH "Ayuda"       XEND
```

The locale.rh header file defines XLAT as a type of resource. XLAT and XEND are delimiters for all the translations of a single string. The same header also defines macros to represent various locale IDs. FRENCH, DUTCH, and ENGLISH_UK, for example, each represent a different LCID. UK is a subdialect of ENGLISH.

Each line in the localization table begins with a resource identifier. These examples use the original string itself to identify the resource that holds its translations.

A localization table is not obliged to provide the same set of translations for each string. For example, it is legal to provide FRENCH_FRANCE, FRENCH_BELGIUM, and

SWEDISH for one string, but only FRENCH and ITALIAN for the next string. Also, if several languages happen to use the same string, it is legal to write the string only once, as in this example:

```
Center XLAT ENGLISH_UK FRENCH GERMAN "Centre" SPANISH "Centro" XEND
```

In British English, French, and German, "Center" is translated as "Centre." In Spanish, it becomes "Centro." Writing "Centre" only once makes the .EXE file smaller.

## Marking translatable strings in the source code

Composing a resource table is the first step, but ObjectComponents still needs to be told when to use the table you have provided. In the automation definition, mark each translatable string by prefixing it with an exclamation point.

```
EXPOSE_METHOD(Clear, TAutoVoid, "!Clear", "Clear accumulator", HC_TCALC_CLEAR)
```

This line from AutoCalc exposes a class method named *Clear*. *Clear* returns **void**. The third parameter, *!Clear*, gives the external name that controllers see. The initial exclamation point tells ObjectComponents to look in the program's executable file for a localization resource whose identifier is the string *Clear*.

```
Clear XLAT GERMAN "AllesLöschen" XEND
```

The exclamation point prefix also marks *Clear* as the language-neutral form of the string. If an automation controller decides to use the locale ID GERMAN, then ObjectComponents tells it that the exposed property is called *AllesLöschen*. If the controller sets any other locale ID, it receives the neutral form, *Clear*.

Argument names as well as properties and methods can be localized.

```
EXPOSE_METHOD(Button, TAutoBool, "!Button", "Button push sequence", HC_TCALC_BUTTON)
  REQUIRED_ARG( TAutoString, "!Key")
```

In determining what to call both the *Button* method and its one argument, ObjectComponents will search the program's localization resources for *Button* and *Key*.

```
Button      XLAT GERMAN "Schaltfläche" XEND
Key         XLAT GERMAN "Taste" XEND
```

The algorithm that searches for resources is not sensitive to case, and the current implementation of 16-bit Windows does not allow the use of extended characters (such as characters with diacritical marks) in resource names. The strings stored in a resource, however, can use any characters and do preserve their case.

A problem arises in naming your resource if the string contains spaces. Resource identifier strings cannot have spaces. Consider what happens if you try to localize the description string for this property:

```
// illegal: no spaces allowed in resource identifiers
EXPOSE_PROPRW(Caption, TAutoString, "!Caption", "!Window Title", HC_TCALCWINDOW_TITLE)
```

It's a good idea to localize descriptions as well as property names, but "Window title" is not a legal resource identifier. In cases like this, use @ instead of ! as the localization prefix, and follow it with any legal identifier.

```
EXPOSE_PROPRW(Caption, TAutoString, "!Caption", "@Caption_",HC_TCALCWINDOW_TITLE)
```

The @ prefix tells ObjectComponents that the string is *only* a resource identifier and should never be displayed no matter what locale the controller requests. To make the distinction even clearer for programmers reading the code, strings used only as identifiers conventionally end with an underscore, as in *Caption_* .

To make "Window Title" the language-neutral string, do not assign it a locale ID in the localization resource.

```
Caption_ XLAT "Window Title" GERMAN "Fenster-Aufschrift" XEND
```

Now a controller that requests any locale setting other than GERMAN is given the string *Window Title*.

Besides ! and @, there is a third localization prefix: #. The # prefix must be followed by digits that identify a localization resource by number.

```
EXPOSE_PROPRW(Caption, TAutoString, "!Caption", "#10047",HC_TCALCWINDOW_TITLE)
```

This example tells ObjectComponents to look for a resource numbered 10047. This is how the resource should appear in the .RC file:

```
10047 XLAT "Window Title" GERMAN "Fenster-Aufschrift" XEND
```

## Understanding how ObjectComponents uses XLAT resources

The external names in macros like EXPOSE_METHOD and EXPOSE_PROPRW are wrapped in objects of type *TLocaleString*, a localizable substitute for **char*** strings. A *TLocaleString* object contains code that searches a program's executable file for XLAT resources. All access to the XLAT resources is performed by *TLocaleString*.

The *TLocaleString* class is defined in osl/locale.h. You don't need to refer to *TLocaleString* directly. The macros and headers bring it in for you.

*TLocaleString* is very efficient. If the controller is working in the server's native language, then *TLocaleString* realizes the strings in the source code already match the locale and it doesn't waste any time reading resources.

Usually ObjectComponents determines the application's default language by reading the system's locale ID at compile time and storing it in the compiled program. You can override the default by including a line like this in your source code.

```
#include "olenls.h"
TLangId TLocaleString::NativeLangId=MAKELANGID(LANG_ENGLISH,SUBLANG_ENGLISH_US);
```

The olenls.h header holds national language support constants, including the MAKELANGID macro and the language and dialect symbols.

When it must resort to resources, *TLocaleString* does everything it can to minimize the time spent searching for translations. When it finds a string to match the current locale, it caches the string in memory and never has to load it again. That means only the first attempt to use each translated string incurs a performance hit. Subsequent requests are satisfied quickly. Once in memory, the strings are stored in a hash table so no space is wasted on duplicates. If *TLocaleString* fails to find a requested string, it remembers the failure as well and won't try to find the same string a second time.

### Localizing registration strings

The same localization mechanism works with strings your application registers. Some strings, such as the *progid*, should not be localized, but the following list names registration keys that can be localized.

- appname
- debugdesc
- description
- format*n*
- menuname
- permname
- typehelp
- verb*n*

The following excerpt from the AutoCalc registration tables shows where to put the localization prefixes. The *appname*, *description*, and *typehelp* keys are localized.

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,       "{877B6200-7627-101B-B87C-0000C057CE4E}")
  REGDATA(progid,      APP_NAME ".Application")
  REGDATA(appname,     "@AppName")
  REGDATA(description, "@Desc")
  REGDATA(cmdline,     "/Automation")
  REGDATA(typehelp,    "@typehelp")
END_REGISTRATION
```

For information about particular registration keys, see the *ObjectWindows Reference Guide*.

AutoCalc supplies translations for the *appname*, *description*, and *typehelp* strings in its resource script. Here are two of them.

```
Desc          XLAT "Automated Calculator 1.0 Application"
            GERMAN "Automatisierte Taschenrechner-Anwendung 1.0"
              XEND
typehelp      XLAT "autocalc.hlp"
            GERMAN "acalcger.hlp"
              XEND
```

ObjectComponents determines the proper language for registration by examining the system settings at run time, but it is possible to override the system setting with the -Language command-line switch (see Table 20.2 on page 349).

## Exposing collections of objects

ObjectComponents lets an automated object expose collections of various types as object properties. The items in a collection can belong to an array, a linked list, or any other structure that organizes sets of similar items. To expose a collection, you need to expose methods for manipulating it. These methods typically include a counter to show the size of the collection, an iterator to walk through the collection, and a random-access function to retrieve specific items in the collection.

A collection object is an object that returns on request individual items from a set of related items. It implements the methods that manipulate the items. In the AutoCalc sample program, the buttons on the face of the calculator are a set of related, similar objects. AutoCalc defines a new class, *TCalcButtons*, whose methods let a controller ask for individual button objects. The buttons themselves are automated objects, so once a

controller receives a button it can send a push command or change the text the button displays.

xposing collections for automation generally involves three steps.

- Exposing the collection as a property of the parent. How you do this depends on the constructor of the class that manages the collection.
- Implementing an iterator in the collection class.
- Implementing other methods in the collection class.

## Constructing and exposing a collection class

If you are converting an application to support automation, you are likely to find that it does not already have a C++ class to act as a collection object. The items might be simple values, structures, or even system objects represented by handles. You have to create a new C++ class, and you have to expose the class in the parent's automation tables as a property of the parent class. How you expose the collection in the parent's automation declaration table depends on what information the parent passes the collection object to construct it. This section considers several different possible constructors and shows the macros for adding the collection as a property of its parent.

Instances of the collection class are constructed only when a controller requests it. The collection object appears to the controller as a property of the parent class. In AutoCalc, for example, when a controller asks for what is in the *Buttons* property, ObjectComponents creates a *TCalcButtons* object on the fly. The constructor of a collection object must accept a single argument passed from the parent to initialize itself.

Because *TCalcButtons* manages a collection of child windows, its parent passes the handle of the parent window. The constructor looks like this:

```
TCalcButtons(HWND window) : HWnd(window) {}
```

For the handle to be passed to the constructor, the parent must add a line to its automation declaration:

```
// from the automation declaration of the parent class
DECLARE_AUTOCLASS(TCalcWindow)
  AUTODATARO(Buttons, hWnd, TAutoObjectByVal<TCalcButtons>,)
```

*Buttons* is assigned as the internal name of a read-only property whose value is *TCalcWindow::hWnd*. For the data type of this property, the table specifies a new class based on the collection class. *TAutoObjectByVal<T>* causes an instance of *T* to be constructed that persists until all external references to that instance are released (when the exposed object goes out of scope in the automation controller).

*TCalcWindow* must also expose the collection property in its automation definition:

```
// from the automation definition of the parent class
DEFINE_AUTOCLASS(TCalcWindow)
  EXPOSE_PROPRO(Buttons, TCalcButtons, "!Buttons", "@Buttons_", HC_TCALCWINDOW_BUTTONS)
```

When a controller asks for what is stored in the read-only property called *Buttons*, ObjectComponents creates a *TCalcButtons* object and passes *hWnd* to its constructor.

Here are three examples showing other ways a parent class might expose a collection object as one of its properties:

- **Case 1:** *TParent::DocList* points to the head of a linked list of *TDocument* objects. A new class, *TDocumentList*, is created as the collection object. The constructor of *TDocumentList* receives from its parent the head of the linked list:

  ```
  TDocumentList(TDocument*);
  ```

  The automation declaration of *TParent* exposes *DocList* as a read-only property, using the collection class to assign it a type.

  ```
  DECLARE_AUTOCLASS(TParent)
    AUTODATARO(Documents, DocList, TAutoObjectByVal<TDocumentList>,)
  ```

  The automation definition of *TParent* calls the collection *Documents* and says its type is *TDocumentList*.

  ```
  DEFINE_AUTOCLASS(TParent)
    EXPOSE_PROPRO(Documents, TDocumentList, "Documents", "Doc Collection", 270)
  ```

- **Case 2:** *TParent* contains a list. It passes **this** to the collection object, *TList*, which extracts list items by indirection through the parent's pointer. The constructor receives the pointer.

  ```
  TList(TParent* owner)
  ```

  The automation declaration of *TParent* exposes **this** as a read-only property, using the collection class to assign it a type.

  ```
  DECLARE_AUTOCLASS(TParent)
    AUTOTHIS(List, TAutoObjectByVal<TList>,)
  ```

  The automation definition of *TParent* calls the collection *List* and says its type is *TList*.

  ```
  DEFINE_AUTOCLASS(TParent)
    EXPOSE_PROPRO(List, TList, "List", "List of items", 240)
  ```

- **Case 3:** *Elem* is an array of integers, defined as **short** *Elem[COUNT]*. The collection object is *TMyArray*, and the constructor receives from the parent a pointer to *Elem*.

  ```
  TMyArray(short* array)
  ```

  The automation declaration of *TParent* exposes *Elem* as a read-only property, using the collection class to assign it a type.

  ```
  DECLARE_AUTOCLASS(TParent)
    AUTODATARO(MyArray, Elem, TAutoObjectByVal<TMyArray>,)
  ```

  The automation definition of *TParent* calls the collection *Array* and says its type is *TMyArray*.

  ```
  DEFINE_AUTOCLASS(TParent)
    EXPOSE_PROPRO(MyArray, TMyArray, "Array", "Array as collection", 110)
  ```

## Implementing an iterator for the collection

The collection class performs whatever actions you want a controller to be able to perform with the collection. Common collection methods include *Count* and *GetObject*,

which return the number of items in the collection or individual items specified by number. The only methods you need to implement, however, are the constructor and an iterator. You have already seen the constructor. An iterator function walks through the collection and returns successive items on each new call.

The easy way to define an iterator is with the AUTOITERATOR macro, which you add to the declaration table of the collection object.

```
DECLARE_AUTOCLASS(TCalcButtons)
  AUTOITERATOR(int Id, Id = IDC_FIRSTID+1, Id <= IDC_LASTID, Id++,
               TAutoObjectByVal<TCalcButton>(::GetDlgItem(This->HWnd,Id)))
```

The parameters to AUTOITERATOR define the algorithm for enumerating objects in the collection. Each of the five macro arguments represents a code fragment, ordered as in a **for** loop.

**1** Declare state variables for keeping track of loop iterations. For example,

```
int Index;
```

**2** Assign initial values to the state variables. For example,

```
Index = 0;
```

**3** Test a Boolean expression to decide whether to enter the loop. For example,

```
Index < This->Total
```

**4** Modify state variables to prepare for the next iteration. For example,

```
Index++;
```

**5** Retrieve one item from the collection. For example,

```
(This->Array)[Index];
```

Note that the server can return any data type for items—values or objects.

In the AUTOITERATOR parameters, do not use commas except inside parentheses. Semicolons can separate multiple statements, but cannot be used to end a macro argument. As in automated methods, *This* is defined to be the **this** pointer of the enclosing C++ class (here, the collection itself).

AUTOITERATOR puts an iterator in the automation declaration table, but the iterator member must still be exposed in the definition table. Use the EXPOSE_ITERATOR macro.

```
EXPOSE_ITERATOR(TAutoShort, "Array Iterator", HC_ARRAY_ITERATOR)
```

EXPOSE_ITERATOR takes fewer parameters than other EXPOSE_*xxxx* macros do. No internal or external names are supplied. A class can have only one iterator, and the external name is always *_NewEnum*. The first parameter describes the type of the items returned from the iterator.

The automation type describes the type of the items returned from the iterator, in the same manner as a function return. The previous example iterates an array of **short int** values, so its automation data type is *TAutoShort*. (For a list of all the automation data types, see Table 21.2.) The second parameter is the documentation string describing the

iterator property, and the third parameter, which is optional, identifies a context in an .HLP file for more information about the iterator.

**Note** From the external side, a script controller sees the enumerator as a property with the reserved name *_NewEnum* that returns an object supporting the standard OLE interface *IEnumVARIANT*. This interface contains methods to perform iteration. A controller makes use of an iterator in a loop like this one, which is written in Visual Basic for Applications:

```
For Each Thing In Owner.Bunch  ("Thing" is an arbitrary iterator name)
  Thing.Member......           (can access methods and properties)
  Next Thing                   (loops through all items in collection)
```

**Note** The AUTOITERATOR macro generates a nested class definition within the collection class. For complex iterators, you can choose to code the iterator explicitly in C++. Here is an example:

```
class TIterator : public TAutoIterator {
  public:
    ThisClass* This;
    /* declare state variables here as members */
    void Init() {/* loop initialization function body */}
    bool Test() {/* loop entry test function body */}
    void Step() {/* loop iteration function body;}
    void Return(TAutoVal& v) {/* current element return: v = expr */}
    TIterator* Copy() {return new TIterator(*this);}
    TIterator(ThisClass* obj, TServedObject& owner)
                 : This(obj), TAutoIterator(owner) {}
    static TAutoIterator* Build(ObjectPtr obj, TServedObject& owner)
    { return new TIterator((ThisClass*)obj, owner); }
};
friend class TIterator;    // make iterator a friend of the surrounding collection class
```

### Adding other members to the collection class

In addition to exposing an iterator, a collection class by convention exposes a *Count* method to return the number of items in the collection, an *Index* method for random access to members of the collection, and optionally, methods such as *Add* and *Delete* to manage the collection externally. Here, for example, is the complete code for the *TCalcButtons* collection class in AutoCalc:

```
class TCalcButtons {    // class used only temporarily to expose collection
  public:
    TCalcButtons(HWND window) : HWnd(window) {}
    short GetCount() { return IDC_LASTID - IDC_FIRSTID; }
    HWND GetButton(short i) {return ::GetDlgItem(HWnd, i + IDC_FIRSTID+1);}
    HWND HWnd;
  DECLARE_AUTOCLASS(TCalcButtons)
    AUTOFUNC0 (Count, GetCount, short,)
    AUTOFUNC1 (Item,  GetButton, TAutoObjectByVal<TCalcButton>, short,
                        AUTOVALIDATE(Arg1 >= 0 && Arg1 < This->GetCount()) )
```

```
    AUTOITERATOR(int Id, Id = IDC_FIRSTID+1, Id <= IDC_LASTID, Id++,
                 TAutoObjectByVal<TCalcButton>(::GetDlgItem(This->HWnd,Id)))
};

DEFINE_AUTOCLASS(TCalcButtons)
  EXPOSE_PROPRO(Count, TAutoLong, "!Count", "@CountBu_", HC_TCALCBUTTONS_COUNT)
  EXPOSE_ITERATOR(TCalcButton, "Button Iterator", HC_TCALCBUTTONS_ITERATOR)
  EXPOSE_METHOD_ID(0, Item, TCalcButton,"!Item",  "@ItemBu_", 0)
    REQUIRED_ARG(TAutoShort, "!Index")
END_AUTOCLASS(TCalcButtons, tfNormal, "TButtonList", "@TCalcButtons", HC_TCALCBUTTONS)
```

## Creating a type library

A type library is a binary file containing information about an automation server. The information describes the objects, properties, and methods the server supports. It is used by programming tools, such as automation controllers, that call the server. Controllers can query the type library for documentation and help with specific objects. The location of its type library is one of the pieces of information an automation server records in the system's registration database.

ObjectComponents can create a type library for you from information in the server's automation definitions. To make a type library, call the server and set the -TypeLib switch on the command line.

```
myapp -TypeLib
```

This command causes ObjectComponents to create a new file, MYAPP.OLB, in the same directory as MYAPP.EXE. ObjectComponents also records the library's location in the registration database.

The -TypeLib flag also accepts an optional path and file name.

```
myapp -TypeLib = data\mytyplib
```

ObjectComponents places MYTYPLIB.OLB in a subdirectory called DATA under the directory where MYAPP.EXE resides.

You can also make ObjectComponents generate mutliple type libraries in different languages with the -Language switch. This command produces two type libraries, one in German and one in Italian.

```
myapp -Language=10 -TypeLib=italiano -Language=7 -TypeLib=deutsch
```

The number passed to -Language must be hexadecimal digits. The Win32 API defines *80C* as the locale ID for the Belgian dialect of the French language. For this command line to have the effect you want, of course, *myapp* must supply Belgian French strings in its XLAT resources.

For more information about localization, see "Localizing symbol names" on page 397. For more about command line switches, see "Processing the command line" on page 349.

You can also create an .HLP file of online Help to accompany your type library. The Help file documents all the commands the server exposes, explaining what arguments they expect and how to use them. If you have a Help file, be sure to register it using the

*typehelp* and *helpdir* registration keys (explained in the *ObjectWindows Reference Guide*). Use the final parameter of the EXPOSE_*xxxx* macros in the automation definition table to associate Help context IDs with each command. If the automation controller asks for help on a command, OLE launches the Help file automatically.

Chapter

22

# Creating an automation controller

This chapter explains how to send commands to other applications through OLE. Chapter 21 shows how to write an automation server, an application that exposes its internal commands to OLE. An *automation controller* is an application that controls a server's automated objects by sending commands to OLE for other programs to execute.

Writing an automation controller is easier than writing a container, a server, or an automation object because sending commands doesn't require any user interface. You don't need to create any windows or use the Clipboard or draw objects on the screen.

In order to send commands to an OLE object, the automation controller must know the names of methods and properties the object's server exposes to OLE. Generally these names come from the server's type library. The controller uses the names in creating C++ proxy classes whose methods send commands to the server. It's possible to browse through available automation objects at run time and discover what commands they support, but to make use of commands discovered at run time usually requires a scripting language.

## Steps for building an automation controller

These are the coding steps required to make one program control another. The sections that follow explain each step in more detail.

1. Include the automation header files in your source code.
2. Create an object of type *TOleAllocator*.
3. Define a proxy C++ class to represent each OLE object you want to automate. Derive the classes from *TAutoProxy*.
4. Implement command methods in your proxy classes with simple automation macros.
5. Construct the proxy objects and call their methods. ObjectComponents sends the commands through OLE to the automation object.

6 Build the program using the medium or large memory model. Link to the OLE and ObjectComponents libraries.

## Including header files

An automation controller needs to include the following headers:

```
#include <ocf/autodefs.h>
#include <ocf/automacr.h>
```

The autodefs.h header defines automation classes such as *TAutoProxy*. The automacr.h header defines the macros a controller uses to implement proxy class methods.

## Creating a TOleAllocator object

Like automation servers, automation controllers must also create a *TOleAllocator* object to initialize the OLE libraries and (optionally) to give OLE a memory allocation function. To create a *TOleAllocator* object, add this line to your program.

```
TOleAllocator OleAlloc;
```

The constructor for *TOleAllocator* initializes the OLE libraries and its destructor releases them. Create an object of type *TOleAllocator* before you begin OLE operations and be sure the object is not destroyed until all OLE operations have ended. A good place to create the *TOleAllocator* is at the beginning of *WinMain* or *OwlMain*.

## Declaring proxy classes

A proxy class is a C++ stand-in for an automated OLE object. You create a proxy class whose interface corresponds to that of the OLE object. By deriving the proxy class from *TAutoProxy*, you connect it to ObjectComponents. When a *TAutoProxy* object is constructed, it calls an OLE API to request the *IDispatch* interface of the automated object that the proxy represents. When you call a function of the proxy class, the proxy sends the corresponding command to the automation server.

An automation controller declares one proxy class for every type of object it wants to control. In simple cases, a single proxy class might be enough. Controlling a complex application that creates several different kinds of automatable objects requires more proxies. To control a spreadsheet, for example, you might need a proxy application class, a proxy spreadsheet class, and a proxy cell class.

The easiest way to declare and implement proxy classes is with the AutoGen utility. AutoGen reads the server's type library and generates C++ source code for the proxy classes a controller needs to send any commands to the server. Simply compile the generated code into your application, construct proxy objects when you need them, and call their member functions to send commands.

As an example of a proxy class, here is the code that AutoGen generates for the automated class *TCalc* in the AutoCalc sample program. The opening comment shows descriptive information from AutoCalc's entries in the registration database including the value of AutoCalc's *version*, *clsid*, and *description* registration keys. The comments for

individual members show the documentation strings that AutoCalc assigns to each member in its automation definition table, the dispatch ID that ObjectComponents assigned to identify each command, and whether the member is a function or a property.

```
// TKIND_DISPATCH: TCalc 1.2 {877B6207-7627-101B-B87C-0000C057CE4E}\409
// Automated Calculator Class
class TCalc : public TAutoProxy {
  public:
    TCalc() : TAutoProxy(0x409) {}
    // Pending operand
    long GetOperand();     // [id(1), prop r/w]
    void SetOperand(long); // [id(1), prop r/w]
    // Calculator accumulator
    long GetAccumulator();     // [id(0), prop r/w]
    void SetAccumulator(long); // [id(0), prop r/w]
    // Pending operation
    TAutoString GetOp();     // [id(3), prop r/w]
    void SetOp(TAutoString); // [id(3), prop r/w]
    // Evaluate operand, op
    TBool Evaluate(); // [id(4), method]
    // Clear accumulator
    void Clear(); // [id(5), method]
    // Update display
    void Display(); // [id(6), method]
    // Terminate calculator
    void Quit(); // [id(7), method]
    // Button push sequence
    TBool Button(TAutoString Key); // [id(8), method]
    // Calculator window
    void GetWindow(TCalcWindow&); // [id(9), propget]
    // Test of object as arg
    long LookAtWindow(TCalcWindow& Window); // [id(10), method]
    // Array as collection
    void GetArray(TCalcArray&); // [id(11), propget]
    // Application object
    void GetApplication(TCalc&); // [id(12), propget]
};
```

The constructor of an automation proxy class must pass to its base class, *TAutoProxy*, a number representing a locale setting. The locale tells what language the automation controller uses when it sends commands to objects. In the example, the number is 0x409, which is the locale ID for American English. AutoGen chooses this locale by reading the system settings when it runs, but you are free to change it to whatever locale you prefer.

The function members of class *TCalc* each send a different command to the calculator object. Read-write properties get two commands, one for getting the value and one for setting it. *GetOp* and *SetOp*, for example, write and read the value representing the next operation the calculator will perform. Other commands, such as *Display* and *Quit*, make the calculator perform some action.

## Implementing proxy classes

Simply declaring methods doesn't accomplish much, of course. You also have to implement them. Each method must send a command through ObjectComponents to the automated object. Here is part of the implementation code that AutoGen generates for the *TCalc* proxy object. Every method simply calls the same three macros.

```
// TKIND_DISPATCH: TCalc 1.2 {877B6207-7627-101B-B87C-0000C057CE4E}\409 Automated
Calculator Class
TAutoString TCalc::GetOp()
{
  AUTONAMES0("Op")
  AUTOARGS0()
  AUTOCALL_PROP_GET
}
void TCalc::SetOp(TAutoString val)
{
  AUTONAMES0("Op")
  AUTOARGS0()
  AUTOCALL_PROP_SET(val)
}
TBool TCalc::Evaluate()
{
  AUTONAMES0("Evaluate")
  AUTOARGS0()
  AUTOCALL_METHOD_RET
}
void TCalc::Clear()
{
  AUTONAMES0("Clear")
  AUTOARGS0()
  AUTOCALL_METHOD_VOID
}
void TCalc::Display()
{
  AUTONAMES0("Display")
  AUTOARGS0()
  AUTOCALL_METHOD_VOID
}
void TCalc::Quit()
{
  AUTONAMES0("Quit")
  AUTOARGS0()
  AUTOCALL_METHOD_VOID
}
void TCalc::GetWindow(TCalcWindow& obj)
{
  AUTONAMES0("Window")
  AUTOARGS0()
  AUTOCALL_PROP_REF(obj)
}
```

The three macros supply all the code needed for each function. The first two macros, AUTONAMES and AUTOARGS, specify what arguments you want to pass. They are explained in more detail below. None of the methods in the example takes any arguments. The AUTOCALL_*xxxx* macros tell whether the command is a function or a property and what kind of value it returns. Table 22.1 lists all the AUTOCALL_*xxxx* macros.

**Table 22.1** Macros for implementing proxy object member functions

| Macro | Description |
|---|---|
| AUTOCALL_METHOD*n*(id, arg...) | Calls a method with *n* arguments that returns a value. |
| AUTOCALL_METHOD*n*V(id, arg...) | Calls a method with *n* arguments that returns **void.** |
| AUTOCALL_METHOD*n*_REF(id, prx, arg...) | Calls a method with *n* arguments that returns a proxy object. |
| AUTOCALL_PROPGET(id) | Retrieves the value of a property. |
| AUTOCALL_PROPSET(id, arg) | Assigns a value to a property. |
| AUTOCALL_PROPREF(id, obj) | Retrieves the value of a property that contains an object. (Objects must be passed by reference.) |

**Note** When an automation command passes an object as a parameter or a return value, be sure to pass by reference, not by assignment. For example, access functions for a property implemented as an object should follow this form:

```
GetObjectX( X& obj );
SetObjectX( X& obj );
```

Passing objects by assignment makes it impossible to provide C++ type safety.

## Specifying arguments in a proxy method

The first two macros in the implementation of a proxy method indicate what arguments you intend to pass. The server can decide that some arguments to a method are optional. You must pass all required arguments, and you can choose to pass any subset of the optional arguments.

For example, a server might expose a method that takes ten arguments, of which five are optional. Optional arguments have default values. Your controller might have a use for only one of the optional arguments, always accepting the default values for the other four. In that case, you can set up your proxy implementation so that you have to pass only six arguments instead of ten.

The AUTONAMES macro lists any optional arguments that you do want to use. It lists them by the names the server assigns to them. (AutoGen reads the names from the server's type library for you.) If you intend to pass only one of five optional arguments, then you list only one argument in AUTONAMES.

The first argument passed to an AUTONAMES macro always indentifies the automation method that this proxy command invokes. The names of arguments come after. If the automation server uses ObjectComponents, then the names used in AUTONAMES come from the server's automation definition table. The function name is the external name in an EXPOSE_METHOD macro, and the argument names come

from subsequent OPTIONAL_ARG macros. (See "Writing definition macros" on page 389.)

The second parameter in a proxy method implementation, AUTOARGS, lists all the arguments that the controller chooses to pass for this command. It tells what will be pushed onto the command stack. AUTOARGS must always list all the required arguments in order first. At the end of the list come any optional arguments from the AUTONAMES macro. If there are five required arguments and the controller wants to pass only one of five optional arguments, then the list in AUTOARGS includes six arguments, the optional one last.

The names used for required arguments are just dummy names. Their position in the list indicates which argument they represent. The names for optional arguments must be the same names used in AUTONAMES. For optional arguments, the name itself is what identifies a particular parameter.

## Creating and using proxy objects

Through a proxy class you can talk to an OLE object, but first the object has to exist. The *TAutoProxy* class defines a member function called *Bind* that asks OLE to create an object. The parameter passed to *Bind* determines the type of object to create. The most convenient identifier is usually a name the automation object has recorded in the registration database. (The object's unique *clsid* number also works but is harder to remember and write.) This is what an automation controller does to make OLE create a calculator object:

```
TCalc calc;                           // create proxy object
calculator.Bind("Calc.Application");  // make OLE create real object
```

The string passed to *Bind* is what the automation server registered as its *progid*:

```
REGDATA(progid, "Calc.Application")    // from server's registration table
```

The destructor for *TAutoProxy* calls the *Unbind* method, so when *calculator* goes out of scope, OLE destroys the actual calculator object.

While *calculator* remains in scope, the controller program issues commands by calling methods on the proxy object. The commands in the following example add 1234 + 4321 and display the result in the calculator's window.

```
calc.SetOperand(1234);
calc.SetOp("Add");
calc.Evaluate();
calc.SetOperand(4321);
calc.Button("+");
calc.Evaluate();
calc.Display();
```

## Compiling and linking

Automation servers and controllers must be compiled with the medium or large memory model. (They run faster in medium model.) They must be linked with the OLE and ObjectComponents libraries.

The integrated development environment (IDE) chooses the right build options for you when you ask for OLE support. To build any ObjectComponents program from the command line, create a short makefile that includes the OCFMAKE.GEN file found in the EXAMPLES subdirectory.

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj myprogram.obj
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES and OBJRES hold the name of the file to build and the names of the object files to build it from. The last line includes the OCFMAKE.GEN file. Name your file MAKEFILE and type this at the command line prompt:

```
make MODEL=m
```

MAKE, using instructions in OCFMAKE.GEN, will build a new makefile tailored to your project. The new makefile is called WIN16M*xx*.MAK.

For more information about OCFMAKE.GEN and the libraries needed for an ObjectComponents program, see page 314.

# Enumerating automated collections

Many automated objects have properties that represent a set of related items—for example, integers in an array, structures in a linked list, or a group of objects such as the buttons on the face of the calculator. To expose a collection, the automation server must implement a collection object with access functions. Chapter 21 explains how an ObjectComponents server defines a C++ collection class to automate its collection (see page 400). As OLE sees it, a collection object implements the standard *IEnumVARIANT* interface. This section explains what a controller must do to use a collection object and enumerate items in the server.

Enumerating a collection takes four steps:

1. Declaring a proxy class for the collection object
2. Implementing the proxy class for the collection object
3. Using the proxy class to declare a collection property
4. Using the proxy class to retrieve the collection and send it commands

## Declaring a proxy collection class

A proxy collection class usually supplies member functions to find out how many items are in the collection, to retrieve individual items randomly by their position in the list, and to enumerate the items in the list sequentially. (On the server's side, ObjectComponents calls this *iterating*. The controller uses the server's iterator to enumerate the items.)

Here is the proxy class that AutoGen creates to enumerate the collection of calculator buttons in AutoCalc.

```
// TKIND_DISPATCH: TButtonList 1.2 {877B6204-7627-101B-B87C-0000C057CE4E}\409
// Button Collection
```

```
class TButtonList : public TAutoProxy {
  public:
    TButtonList() : TAutoProxy(0x409) {}
    // Button Count
    long GetCount(); // [id(1), propget]
    // Button Iterator
    void Enumerate(TAutoEnumerator<TCalcButton>&); // [id(-4), propget]
    // Button Collection Item
    void Item(TCalcButton&, short Index); // [id(0), method]
};
```

The only thing here that wasn't in the previous proxy classes is the use of the *TAutoEnumerator* template. *TAutoEnumerator* encapsulates the code for manipulating the *IEnumVARIANT* interface of a collection object. The type you pass to the template is the type of value the collection contains. In the example, *TCalcButton* is another proxy class representing an automated button object in the server.

## Implementing the proxy collection class

This is the code that AutoGen writes to implement the proxy collection class. *Count* and *Item* are straightforward. The *Enumerate* method does several new things, however.

```
// TKIND_DISPATCH: TButtonList 1.2 {877B6204-7627-101B-B87C-0000C057CE4E}\409 Button
Collection
long TButtonList::GetCount()
{
  AUTONAMES0("Count")
  AUTOARGS0()
  AUTOCALL_PROP_GET
}
void TButtonList::Enumerate(TAutoEnumerator<TCalcButton>& obj)
{
  AUTONAMES0(DISPID_NEWENUM)
  AUTOARGS0()
  AUTOCALL_PROP_REF(obj)
}
void TButtonList::Item(TCalcButton& obj, short Index)
{
  AUTONAMES0("Item")
  AUTOARGS1(Index)
  AUTOCALL_METHOD_REF(obj)
}
```

First, the parameter to the *Enumerate* method is a reference to an object of the type that the collection contains. On successive calls, *Enumerate* returns collection items through this parameter. The data type for the parameter must use the *TAutoEnumerator* template.

Second, the method is identified to AUTONAMES0 as DISPID_NEWENUM. This is a predefined constant from oleauto.h representing the standard dispatch ID (which happens to be –4) for an enumerating command. The AUTONAMES0 macro accepts a dispatch ID instead of a function name. (The other AUTONAMES macros, those that expect argument names as well, require a name string for the function.)

Finally, an enumerator is a property of its object and it passes an object by value, so it the enumerator implementation ends with the AUTOCALL_PROP_REF macro.

## Declaring a collection property

*TButtonsList* is now fully defined as a proxy class for the server's collection object. What's needed now is a way to ask the controller for the collection. In AutoCalc, the collection of buttons is a property of the calculator's automated window object.

```
class TCalcWindow : public TAutoProxy {
  public:
    TCalcWindow() : TAutoProxy(0x409) {}
    // Button Collection
    void GetButtons(TButtonList&); // [id(5), propget]
```

The window class exposes the collection through a *GetButtons* command that returns the value of the collection property. *GetButtons* needs the *TButtonList* class to declare its parameter type.

## Sending commands to the collection

This code from the sample program CallCalc sends the calculator commands that press its buttons. In the code, *window* is the automated window object. *TCalcButton* is the proxy class for individual buttons. *TButtonList* is the proxy object for the collection.

```
TButtonList buttons;                     // declare a collection object
window.GetButtons(buttons);              // bind buttons to automated collection object
TAutoEnumerator<TCalcButton> list;       // create an enumerator of TCalcButton objects
buttons.Enumerate(list);                 // bind list to the server's iterator
TCalcButton button;                      // declare a button object

// list.Step advances to the next item in list
for (i = IDC_FIRSTBUTTON; list.Step(); i++) {
  list.Object(button);                   // bind button to an automated button object
  button.SetActivate(true);              // press the calculator button
}
```

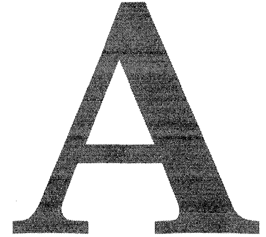
The *buttons*, *list*, and *button* variables are each created in one step and then bound to a server object in another. Each of them is a proxy object for something the server created; *buttons*, for example, is the proxy object for a collection of automated button objects. (See page 401 for an explanation of the server's collection object.) Simply declaring a proxy object, however, does not attach it to any particular automated object in the server. To be able to send commands, a proxy object must be bound to something with an automation interface (*IDispatch* or *IEnumVARIANT*). Because the server defines the collection of buttons as a property of the calculator's window, this command retrieves the collection and connects it to the buttons proxy object:

```
window.GetButtons(buttons);              // bind buttons to automated collection object
```

The two other lines where the comments indicate binding takes place similarly connect *list* to the collection's iterator object and *button* to individual button objects in the collection.

A simple assignment statement might seem more intuitive than the binding step, but the only value that could be assigned in these cases is simply a pointer to an automation interface. A pointer carries no type information; a pointer to a collection's *IDispatch* looks just like the pointer to a button's *IDispatch*. Binding to an existing C++ object preserves information about what kind of automation object it represents.

Appendix



# Converting ObjectWindows code

ObjectWindows 2.5 is a powerful new implementation of the ObjectWindows class library. This version delivers many of the features requested by ObjectWindows 1.0 users:

- Greater type safety
- ANSI C++ compliance
- Support for multiple inheritance
- Automated message cracking
- Broader encapsulations of the Windows API, including support for GDI
- Several new high-level objects, including encapsulations of toolbar and status line functionality
- Transparent targeting of 16-bit and 32-bit applications for Windows NT, Win32s, and Windows 3.1 from a single source code base

To facilitate these new features, there have been several changes to the ObjectWindows class hierarchy. If you have developed applications using ObjectWindows 1.0, this chapter helps you easily convert your existing code base over to ObjectWindows 2.5 so that you can take advantage of the new functionality. In addition, we have provided a utility called OWLCVT that automates the most common changes you may have to make. You can use OWLCVT from the command-line for makefile-based development or from within the IDE if you use project files.

**Note** The term ObjectWindows 1.0 refers to the 1.0*x* version of the ObjectWindows class library, which was provided with the Borland C++ 3.1 and Application Frameworks package and Turbo C++ for Windows 3.1.

The number of changes your code requires depends on which ObjectWindows 1.0 features you've used in your particular application. Although there are some changes that must be made to *any* ObjectWindows 1.0 program, most changes need to be made only if you've used a particular feature. Use the checklist provided in the "Conversion

checklist" section of this chapter to quickly determine which areas of your code are affected.

This chapter is organized into four parts:

- The "Converting your code" section explains the use of the OWLCVT tool, including command-line syntax, how to use it from the IDE, and how OWLCVT modifies your code.
- The "Conversion checklist" section describes all the changes you might have to make to your applications. Along with each description is a page reference telling you what page to turn to for more information about the required change. This lets you read about only those changes you need to make, and ignore those changes that don't apply to your application.
- The "Conversion procedures" section contains detailed technical descriptions of all the changes you might have to make to your applications.
- The "Troubleshooting" section lists a number of common problems you might encounter while converting your code from ObjectWindows 1.0 to ObjectWindows 2.5.

# Converting your code

There are several main steps you must go through to port your ObjectWindows 1.0 code to work with the ObjectWindows 2.5 class library:

1. Make sure your code compiles properly with Borland C++ 4.5. You don't need to be able to link or execute your code; you just need to be able to compile without errors or warnings.
2. Convert your code using the OWLCVT utility.
3. Make any manual conversions needed.

This section discusses these steps and the tools required to do them.

## Converting to Borland C++ 4.5

Before attempting to convert your code, you must make sure it compiles correctly with the Borland C++ 4.5 compiler. Changes to the draft ANSI C++ standard, including the addition of three distinct **char** types and a new syntax for using the **new** and **delete** operators to allocate arrays of objects, could cause your code not to compile. These language changes, and how to fix the problems associated with them, are discussed in the README.TXT file in the section titled "C/C++ Language Changes."

You must also make your code STRICT compliant. Windows 3.1 introduced support in WINDOWS.H for defining STRICT. This enables strict compiler error checking. Code written with STRICT defined is easier to port across platforms and from 16- to 32-bit Windows. You can find more information on making your code STRICT compliant in Chapter 6 in the *Borland C++ Programmer's Guide*.

You can use your existing project files, makefiles, configuration files, response files, and so on, for the compiling process. *Configuration files* are files containing a number of command-line compiler options. *Response files* are files containing both command-line compiler options and file names. Configuration files and response files are discussed in detail in Chapter 3 in the *Borland C++ User's Guide*. The only changes you need to make to your files for this purpose are:

- Change the header file include paths. To properly define ObjectWindows 1.0 classes and ObjectWindows 1.0-compatible container classes, you need to make the following changes:
  - Change C:*BC31*\OWL\INCLUDE to C:*BC45*\INCLUDE\OWLCVT
  - Change C:*BC31*\CLASSLIB\INCLUDE to C:*BC45*\INCLUDE\CLASSLIB\OBSOLETE
  - Change C:*BC31*\INCLUDE to C:*BC45*\INCLUDE

  This assumes the existing paths in your ObjectWindows 1.0-compatible files use the directory C:*BC31*\ as the root directory of your old Borland C++ installation, and that you've installed Borland C++ 4.5 in the directory C:*BC45*. Change these names to reflect the actual directories in which you have your compilers installed.
- Your include paths should be in this order:
  - C:*BC45*\INCLUDE\OWLCVT
  - C:*BC45*\INCLUDE\CLASSLIB\OBSOLETE
  - C:*BC45*\INCLUDE
- Because you only need to make sure your code *compiles* with Borland C++ 4.5, you should remove all linking commands from your makefile or script:
  - If you have explicit linking commands you can either delete them, comment them out, or, if you're using MAKE, specify the appropriate .OBJ files as targets on the MAKE command line.
  - If you're using the compiler to automatically invoke the linker for you, add the **–c** option (to suppress automatically invoking the linker) to your compiler commands.

**Note** If you are using the IDE, select the CPP nodes of your application in the Project window and select Build node from the Project window's SpeedMenu.

If you get any compiler errors or warning messages when you compile your code, correct the problems and recompile. Once your code compiles cleanly, you are ready to move on to converting your code to ObjectWindows 2.5.

## OWLCVT conversions

OWLCVT is a command-line tool you can use to convert your existing ObjectWindows 1.0 code to use the new ObjectWindows 2.5 class libraries. It performs a number of conversions on your ObjectWindows 1.0-compatible source and header files:

- Makes backup copies of any original source or header files that are modified by OWLCVT. See the section "Backing up your old source files."

- Changes the event-handling mechanism from DDVTs to event response tables. See page 424.
- Changes calls to the *TWindowsObject/TWindow* hierarchy to calls to the *TWindow/TFrameWindow* hierarchy. See page 431.
- Preserves calls to native Windows API functions. See page 432.
- Includes the appropriate header files for ObjectWindows 2.5 source. See page 433.
- Includes the appropriate header files for ObjectWindows 2.5 resources. See page 434.
- Replaces calls to *DefWndProc*, *DefCommandProc*, *DefChildProc*, and *DefNotificationProc* with a call to the function *DefaultProcessing*. See page 443.

OWLCVT also inserts comments in your code when it encounters a questionable construct that you might need to modify. You should look for these messages in your converted source files.

## OWLCVT command-line syntax

The command-line syntax for OWLCVT is:

```
OWLCVT [options] file1 [file2 [file3 [...]]]
```

where *filen* is one or more ObjectWindows 1.0 source code files and *options* is one or more command-line compiler options. OWLCVT accepts all regular command-line compiler (BCC.EXE) options. This lets you use any of your old command scripts, makefiles, configuration files, and so on when converting. Only a few of these options have any functional effect on OWLCVT itself, but some options cause macros to be defined in the Borland C++ header files, so you should continue to use the same option sets for converting your files that you used to compile them.

## Backing up your old source files

When you run OWLCVT, it makes a directory called OWLBACK in your current directory. It then makes a copy of your original source file and any local headers and places these in the OWLBACK directory. When OWLCVT has finished converting your files, the modified source files are in your current directory. If, for some reason, the converted files don't function correctly, are corrupted, or are otherwise unsatisfactory, you can easily restore your original files by copying them from the OWLBACK directory. If you run OWLCVT again, and it finds a copy of a file already in the OWLBACK directory, it leaves the copy that's already in the directory and does not overwrite it.

## How to use OWLCVT from the command line

To convert your code from the command line using OWLCVT, follow these steps:

1. Copy the file that contains the compiler options you used for your ObjectWindows 1.0 compilations, such as your makefile, configuration file, response file, and so on, to a new file.

2 Make the following changes to the new file:

- If you haven't already changed the header-file include paths when converting to Borland C++ 4.5, change the include path as follows:
  - Change C:*BC31*\OWL\INCLUDE to C:*BC45*\INCLUDE\OWLCVT (for ObjectWindows 1.0-compatible header files)
  - Change C:*BC31*\CLASSLIB\INCLUDE to C:*BC45*\INCLUDE\CLASSLIB\OBSOLETE (for *Object*-based container class header files)
  - Change C:*BC31*\INCLUDE to C:*BC45*\INCLUDE (for standard header files)

  This assumes the existing paths in your ObjectWindows 1.0-compatible files use the directory C:*BC31* as the root directory of your old Borland C++ installation, and that you have installed Borland C++ 4.5 in the directory C:*BC45*. Change these names to reflect the actual directories in which you have your compilers installed.

**Warning!** If you use any header files that duplicate the names of Borland header files, you *must* place the directory containing these files in your header file include path *before* the Borland include directories. You must do this even if the files are in the current directory and you use the **#include "*filename*.h"** syntax to include these files.

- If you're using a makefile, batch file, or any type of command script:
  - Remove all commands except for C++ compilations, including linking, resource compiling and binding, and so on. For example, suppose you have the following batch file:

    ```
    BCC -WS -c -ml -w MYAPP.CPP
    RC -r -iC:\BC31\OWL\INCLUDE -iC:\BC31\INCLUDE MYAPP.RC
    TLINK  /Tw /c /C COWL MYAPP, MYAPP, , @MAKE0000.$$$, MYAPP.DEF
    ```

    This assumes that your existing files refer to a compiler in the directory C:*BC31*. Change this to reflect the actual directory in which you have your old compiler installed. After removing all commands except for C++ compilations, this file would look like this:

    ```
    BCC -WS -c -ml -w MYAPP.CPP
    ```

  - Convert the compilation commands into OWLCVT commands. For example, suppose you had converted the batch file in the previous step. After converting the compilation command into an OWLCVT command, this file would look like this:

    ```
    OWLCVT -WS -c -ml -w MYAPP.CPP
    ```

3 Run the appropriate command-line tool. For example, if you're using a batch file, run the batch file; if you're using a makefile, run MAKE, and so on. If you're using a configuration file or response file from the command line, run OWLCVT just like you would the compiler. For example, if you had the file configuration file MYCONVRT.CFG, and you wanted to convert the file MYFILE.CPP, the OWLCVT command line would look like this:

```
OWLCVT +MYCONVRT.CFG MYFILE.CPP
```

4 Once all your files have been processed by OWLCVT, you should check whether any further modifications are necessary. These changes are discussed in the next section.

5. Once you have made any manual changes necessary, build your project using the Borland C++ 4.5 tools. You also need to restore resource-compilation commands in your makefile. Note that RC.EXE has been replaced in Borland C++ 4.5 with BRC.EXE, the Borland Resource Compiler. Explicit calls to TLINK also need to be restored and updated to use new startup code and libraries supplied by Borland C++ 4.5.

## How to use OWLCVT in the IDE

To convert your code from the IDE using OWLCVT, follow these steps:

1. Load your project file into the IDE by using Project | Open project. The IDE will automatically make the necessary library changes in TargetExpert for your conversion to OWL 2.5.
2. If you haven't already changed the header-file include paths when converting to Borland C++ 4.5, make the following changes under Options | Project | Directories:
   - Change C:*BC31*\OWL\INCLUDE to C:*BC45*\INCLUDE\OWLCVT (for ObjectWindows 1.0-compatible header files)
   - Change C:*BC31*\CLASSLIB\INCLUDE to C:*BC45*\INCLUDE\CLASSLIB\OBSOLETE (for *Object*-based container class header files)
   - Change C:*BC31*\INCLUDE to C:*BC45*\INCLUDE (for standard header files)

   This assumes the existing paths in your ObjectWindows 1.0-compatible files use the directory C:*BC31* as the root directory of your old Borland C++ installation, and that you have installed Borland C++ 4.5 in the directory C:*BC45*. Change these names to reflect the actual directories in which you have your compilers installed.

   **Warning!** If you use any header files that duplicate the names of Borland header files, you *must* place the directory containing these files in your header file include path *before* the Borland include directories. You must do this even if the files are in the current directory and you use the **#include "*filename*.h"** syntax to include these files.
3. Select the CPP nodes of your application in the Project window, click your right mouse button, and select Special | OWL Convert from the Project window's SpeedMenu. The IDE automatically passes the command-line options from your project to OWLCVT along with the file names of your selected nodes. If OWL Convert does not appear under Special on the Project window's SpeedMenu, you must install it under Options | Tools.
4. Once all your files have been processed by OWLCVT, you should check whether any further modifications are necessary. These changes are discussed in the next section.
5. Once you have made any manual changes necessary, build your project using the Borland C++ 4.5 tools.

# Conversion checklist

This section presents a number of conversions that you might need to make to your existing ObjectWindows 1.0 code after running OWLCVT. Most of these conversions

are necessary only if you use a particular feature of ObjectWindows 1.0. OWLCVT also performs a number of conversions automatically (see page 419). The following conversions need to be done manually, but *only* if you use that particular feature or class:

- **Constructing virtual bases**: A number of classes have been modified in ObjectWindows 2.5 to use virtual base classes. See page 435.
- **Downcasting virtual bases to derived types**: To downcast a virtual base class pointer to a derived class (for example, passing a *TWindow* * in place of a *TFrameWindow* *), use the TYPESAFE_DOWNCAST macro. See page 435.
- **Moving from Object-based containers to the BIDS library**: The *Object*-based container class library isn't used in ObjectWindows 2.5. See page 436.
- **Streaming**: There have been a number of changes to the streams library. See page 437.
- **MDI classes**: There are a number of changes you need to make when using the *TMDIFrame* and *TMDIClient* classes. See page 438.
- **MainWindow variable**: You should no longer set the variable *TApplication::MainWindow*. Instead you should use the *SetMainWindow* function. See page 441.
- **Using a dialog as the main window**: There are a number of changes you need to make if you're using a dialog as your main window. See page 441.
- **TApplication message processing functions**: The *ProcessDlgMsg*, *ProcessAccels*, and *ProcessMDIAccels* functions have been removed from the *TApplication* class. See page 442.
- **Paint function**: The declaration for the *TWindow* member function *Paint* has changed. See page 444.
- **CloseWindow, ShutDownWindow, and Destroy functions**: The declarations for these *TWindow* member functions has changed. See page 445.
- **ForEach and FirstThat functions**: The declarations for the *TWindow* member functions *ForEach* and *FirstThat* have changed. See page 445.
- **TComboBoxData and TListBoxData classes**: Some data members of *TListBoxData* and *TComboBoxData* classes have changed type. See page 446.
- **TEditWindow and TFileWindow classes**: *TEditWindow* and *TFileWindow* have been replaced by *TEditSearch* and *TEditFile*. See page 446.
- **TSearchDialog and TFileDialog classes**: The *TSearchDialog* and *TFileDialog* classes have been replaced by the *TReplaceDialog* or *TFindDialog* and the *TFileOpenDialog* classes. See page 448.
- **ActivationResponse function**: The *ActivationResponse* function has been removed from the *TWindow* and *TWindowsObject* classes. Examples of how to attain the same functionality in ObjectWindows 2.5 are given on page 448.

- **BeforeDispatchHandler and AfterDispatchHandler functions**: The *BeforeDispatchHandler* and *AfterDispatchHandler* functions have been removed from ObjectWindows. Examples of how to attain the same functionality in ObjectWindows 2.5 are given on page 448.
- **DispatchAMessage function**: The *DispatchAMessage* function has been removed from ObjectWindows. See page 449.
- **KBHandlerWnd data member**: The KBHandlerWnd data member has been removed from the *TApplication* class. See page 449.
- **MAXPATH**: MAXPATH is no longer defined in any ObjectWindows header files. It is now defined only in the header file dir.h. See page 450.
- **Style conventions**: ObjectWindows 2.5 uses somewhat different style conventions from ObjectWindows 1.0. Although your application should compile fine without these stylistic changes, you should make these changes anyway to ensure easy compatibility with your future ObjectWindows 2.5 code. See page 450.

# Conversion procedures

This section contains detailed technical descriptions of the procedures outlined in the two previous sections.

## Handling messages and events

DDVTs (dynamic dispatch virtual tables), which ObjectWindows 1.0 uses to handle application events, have some limitations, especially with multiple inheritance and 32-bit environments. ObjectWindows 2.5 replaces DDVTs with *event response tables*, which offer the following advantages over DDVTs:

- Full support for multiple inheritance of window classes
- Automated message cracking
- Compile-time type checking of all event-handling functions and cracked message parameters
- Compatibility between 16-bit and 32-bit environments
- Easier use of user-defined and run-time-defined messages
- Ability to dispatch two or more messages to a single event-handling function
- Full compliance with the draft ANSI C++ standard

OWLCVT automatically converts your existing DDVTs into ObjectWindows 2.5 response tables. OWLCVT does *not* maintain your symbolic constants, and instead converts them to their numeric values. For example, suppose you have the following DDVT declaration:

```
virtual void CMTest(TMessage& Msg) = [CM_FIRST + CM_TEST];
```

When OWLCVT converts this, it uses the numeric value of the defined CM_TEST:

```
DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
  EV_COMMAND(101, CMTest),
END_RESPONSE_TABLE;
```

The following sections describe how to convert DDVTs to response tables manually. However, it is not recommended that you try to do this task manually, especially for a large application.

**Note** The following sections only describe how to convert your existing ObjectWindows DDVTs. Response tables offer more features you'll probably want to take advantage of. For complete details about event response tables, see Chapter 4.

Creating event response tables consists of four steps, which the following sections describe:

1. Removing DDVT functions
2. Adding an event response table declaration
3. Adding an event response table definition
4. Adding event response table entries

## Removing DDVT functions

You should first remove the DDVT function declarations from your window class definition. You need to remove the DDVT dispatch index (for example, CM_FIRST + CM_SENDTEXT), since the member function definition doesn't use it. The second part of the dispatch index is used when you define your response table. You can also remove the **virtual** keyword because event response tables don't require event response functions to be virtual.

Here are some DDVT function declarations and their event response table equivalents:

| ObjectWindows 1.0 | ObjectWindows 2.5 |
|---|---|
| virtual void CMSendText(TMessage &Msg) = [CM_FIRST + CM_SENDTEXT]; | void CmSendText(); |
| virtual void CMEmpInput(TMessage &Msg) = [CM_FIRST + CM_EMPINPUT]; | void CmEmpInput(); |
| virtual void HandleListBoxMsg(TMessage &Msg) = [ID_FIRST + ID_LISTBOX]; | void HandleListBoxMsg(uint); |
| virtual void WMInitMenu(RTMessage) = [WM_FIRST + WM_INITMENU]; | void EvInitMenu(WPARAM); |
| virtual void BNClicked(RTMessage Msg) = [NF_FIRST + BN_CLICKED]; | void BNClicked(); |

Each predefined Windows message has a specific message-handling function associated with it. In addition, each function has a specific signature that you must use when writing your own code for handling these messages. The Windows messages and their corresponding function names and signatures are listed in Chapter 3 of the *ObjectWindows Reference Guide*.

If you use custom Windows messages, the function name is up to you. You specify the function name using one of the response table macros described in Table 22.2. The function signature depends on which macro you use. See the *ObjectWindows Reference Guide* for more information.

### Naming conventions

You should name ObjectWindows 2.5 event-handling functions by prefixing the name of the function with two letters taken from the message type (such WM, EV, CM, and so on). The first letter should be uppercase and the second letter should be lowercase; don't use two uppercase letters. For example, *CMCommand* becomes *CmCommand*. The predefined ObjectWindows message-handling functions are all named according to this style.

OWLCVT converts ObjectWindows-1.0 style function names to the ObjectWindows 2.5 style. If you make a call to the base class version of a function, however, OWLCVT does *not* convert that call. You need to convert these calls manually. For example, suppose your ObjectWindows 1.0 application has a class called *TMyWindow* that has a function *WMSize* that calls the *TWindowsObject::WMSize* function. OWLCVT converts the *TMyWindow::WMSize* function name to *TMyWindow::EvSize* and the base class name from *TWindowsObject* to *TWindow*, but it doesn't convert the call to the base class *WMSize* function. You need to convert this name to *EvSize* manually.

## Adding an event response table declaration

The next step is to add an event response table declaration after the last declaration in your window class. For example:

```
class TMyWindow: public TFrameWindow
{
  ⋮
  DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

DECLARE_RESPONSE_TABLE is a macro that takes the name of the class as its parameter. See Chapter 4 for more details about event response table declarations.

## Adding an event response table definition

In conjunction with the DECLARE_RESPONSE_TABLE macro, you need to add an event response table definition in the source file (*not* a header file) where you define the members of your window class. You also need to add event response table entries, which the following sections discuss. Here's a sample event response table definition:

```
// NOTE: Response tables should be defined in global scope.
DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
  // event response table entries
  ⋮
END_RESPONSE_TABLE;
```

DEFINE_RESPONSE_TABLE*X* is a macro that takes the name of the window class and its immediate base classes as its parameters. The *X* is based on the number of base classes your class has. END_RESPONSE_TABLE is a macro that ends the event response table definition. See Chapter 4 for more information about defining event response tables.

## Adding event response table entries

In ObjectWindows 1.0, the dispatch index you used in a message response member function's declaration determined what kind of message the function responded to. For example, the CM_FIRST constant identified command response member functions.

ObjectWindows 2.5's event response tables offer all of ObjectWindows 1.0's dispatch types and several more. Table 22.2 lists the ObjectWindows 1.0 dispatch types and their ObjectWindows 2.5 event response table equivalents. See the following sections for information specific to each dispatch type.

**Table 22.2** Message response member functions and event response tables

| Type of message response function | Version 1.0 constant | Version 2.5 response table entry |
|---|---|---|
| Command message | CM_FIRST | EV_COMMAND |
| Child ID-based message | ID_FIRST | EV_CHILD_NOTIFY_ALL_CODES |
| Notify-based message | NF_FIRST | EV_NOTIFY_AT_CHILD |
| Windows messages | WM_FIRST | EV_MESSAGE and EV_WM_XXX |

### Responding to command messages

Command messages are those for which Windows sends a WM_COMMAND message from a menu or accelerator. In ObjectWindows 1.0, you'd declare a member function using the sum of CM_FIRST and the menu or accelerator resource ID; ObjectWindows intercepted the WM_COMMAND message and called the message response member function with the matching ID.

In ObjectWindows 2.5, you do the same thing, but you use event response tables instead of DDVTs. Here's an example:

```
// ObjectWindows 1.0 member function declaration
virtual void CMSendText(TMessage &Msg) = [CM_FIRST + CM_SENDTEXT];

// ObjectWindows 2.5 event response table entry and member function
EV_COMMAND(CM_SENDTEXT, CmSendText),
void CmSendText();
```

### Responding to child ID-based messages

Child ID-based message response member functions handle all the messages coming from a control that ObjectWindows passed along to the control's parent window. In ObjectWindows 1.0, the control notification code was passed in the *TMessage.LP.Hi* member, which the message response member function had to check for, usually with a **switch** statement.

ObjectWindows 2.5 supports the same kind of dispatching with the EV_CHILD_NOTIFY_ALL_CODES event response table entry; all the notification codes are passed to a single member function. Here's an example:

```
// ObjectWindows 1.0 member function declaration
virtual void HandleListBoxMsg(TMessage &Msg) = [ID_FIRST + ID_LISTBOX]

// ObjectWindows 2.5 event response table entry and function definition
```

```
EV_CHILD_NOTIFY_ALL_CODES(ID_LISTBOX, HandleListBoxMsg),
void HandleListBoxMsg(uint);
```

ObjectWindows 2.5 also supports dispatching specific notification codes to specific member functions, something ObjectWindows 1.0 doesn't support. Use the EV_CHILD_NOTIFY event response table entry for such dispatching. Here's an example:

```
EV_CHILD_NOTIFY(ID_BUTTON, HandleButtonClick, BN_CLICKED),
void HandleButtonClick();
```

Since you often need to respond to Windows control notification codes, ObjectWindows defines macros to more easily handle button, combo box, edit control, and list box notification codes. Here's an example that simplifies the LBN_DBLCLK notification code:

```
EV_LBN_DBLCLK(ID_LISTBOX, HandleListBoxMsg),
void HandleListBoxMsg(uint);
```

**Note** Child ID-based messages are actually command messages that include a notification code. For command buttons, the notification code is zero, which makes it look like a menu command message. The recommended way of responding to button presses is with command message response functions rather than child ID-based message response functions. For example, an OK button is usually a child window to a dialog box. When the user clicks it, the button passes a message that can be handled like a command message. You can handle the button message like this:

```
EV_COMMAND(IDOK, CmOk),
```

### Responding to notification messages

Notification messages are like child ID-based messages but instead of being handled by the parent window, they're handled by the control itself. Notification messages are best for creating specialized control classes.

ObjectWindows 1.0 and 2.5 both dispatch notification messages to specific member functions, as this example shows:

```
// ObjectWindows 1.0 member function declaration
virtual void ENChange(TMessage &Msg) = [NF_FIRST + EN_CHANGE]

// ObjectWindows 2.5 event response table entry and function definition
EV_NOTIFY_AT_CHILD(EN_CHANGE, ENChange),
void FNameChange();
```

### Responding to general messages

You can also respond to messages that aren't command messages, child ID-based messages, or notification messages.

ObjectWindows 1.0 and 2.5 dispatch Windows messages to specific member functions. Notice that the ObjectWindows 2.5 naming convention for Windows messages is to use the prefix *Ev* with a mixed-case version of the Windows message constant:

```
// ObjectWindows 1.0 member function declaration
virtual void WMCtlColor(TMessage &Msg) = [WM_FIRST + WM_CTLCOLOR]
```

```
// ObjectWindows 2.5 event response table entry
EV_MESSAGE(WM_CTLCOLOR, EvCtlColor),
```

As with child ID-based messages, ObjectWindows defines macros to make it easy to respond to Windows messages. Here's an example that uses the predefined macro for the WM_CTLCOLOR message:

```
// ObjectWindows 2.5 event response table entry
EV_WM_CTLCOLOR
```

Using the predefined macros assumes you name your event response function using the *Ev* naming convention.

Another good reason to use the predefined macros is that ObjectWindows automatically "cracks" the parameters that are normally passed in the *LPARAM* and *WPARAM* parameters.

For example, using EV_WM_CTLCOLOR assumes that you have an event response member function declared like this:

```
HBRUSH EvCtlColor(HDC hDCChild, HWND hWndChild, uint nCtrlType);
```

Message cracking provides for strict C++ compile-time type checking, which helps you catch errors as you compile your code rather than at run time. See Chapter 4 for more details about the predefined message macros.

## Event response table samples

Here are several ObjectWindows 1.0 window class declarations and their ObjectWindows 2.5 equivalents:

**ObjectWindows 1.0:**

```
class TMyWindow: public TWindow
{
  ⋮
  protected:
    virtual void WMCtlColor(TMessage &Msg) = [WM_FIRST + WM_CTLCOLOR];
    virtual void WMPaint(TMessage &Msg) = [WM_FIRST + WM_PAINT];
    virtual void CMSendText(TMessage &Msg) = [CM_FIRST + CM_SENDTEXT];
    virtual void CMEmpInput(TMessage &Msg) = [CM_FIRST + CM_EMPINPUT];
};
```

**ObjectWindows 2.5:**

```
class TMyWindow: public TFrameWindow
{
  ⋮
  protected:
    LPARAM EvMyMessage(WPARAM, LPARAM);
    void EvPaint();
    void CmSendText();
    void CmEmpInput();

  DECLARE_RESPONSE_TABLE(TMyWindow);
```

```
};

DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
  EV_MESSAGE(WM_MYMESSAGE, EvMyMessage),
  EV_WM_PAINT,
  EV_COMMAND(CM_SENDTEXT,CmSendText),
  EV_COMMAND(CM_EMPINPUT, CmEmpInput),
END_RESPONSE_TABLE;
```

**ObjectWindows 1.0:**

```
class TMyDialog: public TDialog
{
  ⋮
  protected:
    virtual void HandleListBoxMsg(TMessage &Msg) = [ID_FIRST + ID_LISTBOX];
};
```

**ObjectWindows 2.5:**

```
class TMyDialog: public TDialog
{
  ⋮
  protected:
    void HandleListBoxMsg(uint);

  DECLARE_RESPONSE_TABLE(TMyDialog);
};

DEFINE_RESPONSE_TABLE (TMyDialog, TDialog)
  EV_CHILD_NOTIFY_ALL_CODES(ID_LISTBOX, HandleListBoxMsg),
END_RESPONSE_TABLE;
```

**ObjectWindows 1.0:**

```
class TMyButton: public TButton
{
  protected:
    virtual void BNClicked(TMessage &Msg) = [NF_FIRST + BN_CLICKED];
};
```

**ObjectWindows 2.5:**

```
class TMyButton: public TButton
{
  ⋮
  protected:
    void BNClicked ();

  DECLARE_RESPONSE_TABLE(TMyButton);
};

DEFINE_RESPONSE_TABLE(TMyButton, TButton)
  EV_NOTIFY_AT_CHILD(BN_CLICKED, BNClicked),
END_RESPONSE_TABLE;
```

## Changing your window objects

ObjectWindows 1.0 had two classes for "generic" windows: *TWindowsObject* and *TWindow*. *TWindowsObject* was an abstract class; it provided the basic behavior for all windows, dialog boxes, and other interface elements, but an instance of *TWindowsObject* wasn't very useful by itself. *TWindow*, on the other hand, served as the class you used for all types of windows. Unfortunately, that meant that even simple child *TWindow* objects had functionality and code they didn't use.

ObjectWindows 2.5 offers two new classes: *TWindow* and *TFrameWindow*. *TWindow* is similar to *TWindowsObject* in ObjectWindows 1.0, except that it's not abstract. You can use instances of *TWindow* in ObjectWindows 2.5 for child windows. *TFrameWindow* objects serve as overlapped or popup main windows; they maintain a client window, and are inherited by *TMDIFrame* for MDI support and *TDecoratedFrame* for *decoration* support (like tool bars and status bars).

### Converting constructors

OWLCVT performs a search and replace operation on your source files, replacing all occurrences of *TWindow* with *TFrameWindow*, and all occurrences of *TWindowsObject* with *TWindow*. However, this modification isn't sufficient because the *TFrameWindow* constructor does not always take the same parameters as the old *TWindow* constructor. There were two constructors for the ObjectWindows 1.0 *TWindow* class:

```
TWindow(PTWindowsObject, LPSTR, PTModule = NULL);
TWindow(HWND, PTModule = NULL);
```

OWLCVT converts the *TWindow* name to *TFrameWindow*. But after this conversion, neither of these constructors corresponds directly to the available *TFrameWindow* constructors:

```
TFrameWindow(TWindow *parent,
             const char far *title = 0,
             TWindow *clientWnd = 0,
             bool shrinkToClient = false,
             TModule *module = 0);
TFrameWindow(HWND hWnd, TModule *module = 0);
```

However, the two most common usages of the *TWindow* constructor in ObjectWindows 1.0 were as follows:

```
// First TWindow constructor, PTModule parameter set to its default value.
TWindow(AParent, "Title");

// Second TWindow constructor, PTModule parameter set to its default value.
TWindow(AParent);
```

OWLCVT converts these calls to:

```
TFrameWindow(parent, "Title");
TFrameWindow(parent);
```

These calls compile correctly. The first call sets the last three parameters of the five-parameter *TFrameWindow* constructor to their respective defaults. The second call sets the second parameter of the two-parameter *TFrameWindow* constructor to its default.

You shouldn't have to make any further changes unless you determine you need to specify a value for any of the other parameters.

If your ObjectWindows 1.0 code specifies a value for the PTModule parameter, the conversion of your constructor as done by OWLCVT might not correspond to a valid *TFrameWindow* constructor. For example, the *TWindow* constructors might look something like this:

```
TWindow(AParent, ptModule);
TWindow(AParent, "Title", ptModule);
```

The converted code would look like this:

```
TFrameWindow(parent, ptModule);
TFrameWindow(parent, "Title", ptModule);
```

The second call compiles and functions correctly. To make the first call compile correctly, you can remove the ptModule variable entirely, as shown here:

```
TFrameWindow(parent, "Title");
```

This way, the final three parameters of the five-parameter constructor take on their default values. You can also fill in default values for the third and fourth parameters:

```
TFrameWindow(parent, "Title", 0, false, ptModule);
```

Refer to the *ObjectWindows Reference Guide* section on the *TFrameWindow* class to learn more about the *TFrameWindow* constructors and their parameters.

## Calling Windows API functions

ObjectWindows 2.5 encapsulates much more of the Windows API than ObjectWindows 1.0. The advantage of this is that ObjectWindows takes care of passing common parameters, such as window handles, to the API functions. But because some ObjectWindows 2.5 member functions have the same names as Windows API functions, you might get compile-time errors like this:

```
Extra parameter in call to TClass::MessageBox(const char far *, const char far *, unsigned
 int)
```

The easiest way to get your code to work is to use the :: scope resolution operator. For example, suppose you made the following call to the Windows API function *MessageBox* in your ObjectWindows 1.0 application:

```
void
TMyWindow::CMAddRecord()
{
  MessageBox(HWindow, "All fields must be filled in", "Input Error", MB_OK);
}
```

You can force this function to call the Windows API function with ObjectWindows 2.5 by adding the :: scope resolution operator:

```
void
TMyWindow::CMAddRecord()
{
  ::MessageBox(HWindow, "All fields must be filled in", "Input Error", MB_OK);
}
```

You can also use the encapsulated API function *TWindow::MessageBox*:

```
void
TMyWindow::CMAddRecord()
{
  MessageBox("All fields must be filled in", "Input Error", MB_OK);
}
```

The advantage of using the encapsulated ObjectWindows equivalent is that you do not have to pass window parameters explicitly. These are handled by the *TWindow* member functions inherited by the class you're using to make the call. OWLCVT automatically prefixes any calls to Windows API functions with the :: scope resolution operator.

## Changing header files

You need to make these two changes to the way you include some header files in your code:

- Use the new header file locations
- Use the new streamlined ObjectWindows header files

### Using the new header file locations

Borland C++ 4.5 places all header files under the INCLUDE directory. ObjectWindows header files are now in the INCLUDEOWL directory. The header files for the container class library and run-time library are also under the INCLUDE directory.

In versions of Borland C++ prior to 4.5, you might have set your include directories path to something like C:BORLANDC\INCLUDE; C:BORLANDC\OWL\INCLUDE; C:BORLANDC\CLASSLIB\INCLUDE. In Borland C++ 4.5, all you need is C:BORLANDC\INCLUDE. In your code, instead of including header files with directives like `#include <applicat.h>`, you now include ObjectWindows or class library header files like `#include <owl\applicat.h>` or `#include <classlib\arrays.h>`. All of the ObjectWindows source code and sample applications use this approach.

You can also include resource script files and resource header files this way. For example, to include the resource header and resource script files for *TPrinter*, the **#include** statement would look like this:

```
#include <owl\printer.rh>
#include <owl\printer.rc>
```

### Using the new streamlined ObjectWindows header files

ObjectWindows 2.5 header files contain fewer class declarations than their ObjectWindows 1.0 counterparts. Since fewer classes are declared in each file, you probably have to explicitly include more header files. For example, in ObjectWindows 1.0, including owl.h caused several classes to be defined, including *TWindowsObject*,

*TWindow*, *TMDIFrame*, *TMDIClient*, and *TDialog*. The functionality of including owl.h can be achieved by including applicat.h, framewin.h, dialog.h, mdi.h, scroller.h, and dc.h.

**Note** The header file owl\owlpch.h includes all the ObjectWindows header files, which can be useful for creating an ObjectWindows precompiled header file. For example, the following fragment creates a precompiled header file using owl\owlpch.h:

```
#pragma hdrfile "OWLPCH.CSM"
#include <owl\owlpch.h>
#pragma hdrstop
```

The advantage of using precompiled header files is that they provide a great increase in compilation speed, reducing the time it takes to process header files by up to 90%. For more information on precompiled headers, see Chapter 6 in the *Borland C++ Programmer's Guide.*

# ObjectWindows resources

ObjectWindows 1.0 combined the resources and identifiers used by several classes into only a few files. If your application used the resources of one class, you also got the resources for a number of other classes, regardless of whether you used them. ObjectWindows 2.5 provides one resource script file and one resource header file per class (a resource header file contains all the identifiers for the resources defined in the resource script file) for each class that requires resources.

This prevents including resources or header files unnecessarily. The names of the resource script and header files parallel the corresponding header file names. For example, the *TPrinter* class is defined in the header file printer.h. The resource IDs for the *TPrinter* class are contained in the file printer.rh. The resources used by the *TPrinter* class are contained in the file PRINTER.RC.

## Compiling resources

When compiling your resources, you should be sure you modify the header file include path for the resource compiler. The ObjectWindows 1.0 header file include path usually included the directories C:*BC31*\INCLUDE, C:*BC31*\OWL\INCLUDE, and C:*BC31*\CLASSLIB\INCLUDE. For Borland C++ 4.5, this path should be changed to search C:*BC45*\INCLUDE and OWL prefixed on the file name, as shown on page 433.

This assumes the existing paths in your ObjectWindows 1.0-compatible files use the directory C:*BC31* as the root directory of your old Borland C++ installation, and that you have installed Borland C++ 4.5 in the directory C:*BC45*. Change these names to reflect the actual directories in which you have your compilers installed.

To bring in the resources for an ObjectWindows class, just include the appropriate resource file from your own resource script file. For example, to add the resources for the *TPrinter* class, you would add the following line to your own .RC file:

```
#include <owl\printer.rc>
```

### Menu resources

When using menu resources in your code, you might need to change the way menus are assigned to your frame window objects. ObjectWindows 1.0 let you directly assign a menu to a frame window object by setting the *Menu* member of the object's *Attr* structure equal to a particular resource ID. For example:

```
Attr.Menu = MENU_1;
```

ObjectWindows 2.5 doesn't permit this type of assignment. Instead, you should use the *TFrameWindow::AssignMenu* function. The previous line of code looks like this using the *AssignMenu* function:

```
AssignMenu(MENU_1);
```

## Constructing virtual bases

A number of classes that took nonvirtual base classes in ObjectWindows 1.0 are derived from virtual base classes in ObjectWindows 2.5. For the purposes of porting, the classes that are affected by this are classes that use *TWindow* and *TFrameWindow* as virtual bases: *TDialog*, *TMDIFrame*, *TFrameWindow*, *TMDIChild*, *TDecoratedFrame*, *TLayoutWindow*, *TClipboardViewer*, *TKeyboardModeTracker*, and *TTinyCaption*. In C++, virtual base classes are constructed first, which means that the derived class' constructor cannot specify default arguments for the base class constructor. You can find a number of ways to properly initialize virtual bases on page 65.

## Downcasting virtual bases to derived types

A fairly common practice in ObjectWindows 1.0 code is to cast a *TWindowsObject* pointer to a derived type. The *TWindow* base class (the ObjectWindows 2.5 equivalent of *TWindowsObject*; see page 431) is a virtual base in many of the standard ObjectWindows 2.5 classes; however the C++ language doesn't let you downcast a virtual base class pointer to a derived class. To convert this type of construct to ObjectWindows 2.5, you must use the TYPESAFE_DOWNCAST macro. The TYPESAFE_DOWNCAST macro takes two parameters. The first parameter is the data type you want to downcast to. The second parameter is the class instance you want to downcast.

For example, the following code downcasts the *TWindowsObject* object pointer *Parent* to a *TWindow*:

```
void
TMyChildWindow::MyFunc()
{
  ⋮
  // Parent is actually a TWindowsObject object.
  ((TWindow *)Parent)->AssignMenu("NewMenu");
  ⋮
}
```

You might try to convert this code like this, simply converting the *TWindow* class to a *TFrameWindow* class:

```
void
TMyChildWindow::MyFunc()
{ // error on next line
  ⋮
  // Parent is actually a TWindow object.
  ((TFrameWindow *)Parent)->AssignMenu("NewMenu");
  ⋮
}
```

However, in ObjectWindows 2.5, Parent's type is a *TWindow* * (it was a *TWindowsObject* *), which is a virtual base of *TFrameWindow*. Attempting to downcast this results in a compile-time error. The correct way to convert this using the TYPESAFE_DOWNCAST macro is shown here:

```
void
TMyChildWindow::MyFunc()
{
  ⋮
  TFrameWindow* frame = TYPESAFE_DOWNCAST(TFrameWindow*,Parent);
  if(frame)
    frame->AssignMenu("NewMenu");
  ⋮
}
```

Here's the syntax for the TYPESAFE_DOWNCAST macro:

```
type TYPESAFE_DOWNCAST(type, object)
```

where:

- *type* is the data type to which you want to cast the object.
- *object* is the object you want to cast.

If the conversion is successful, TYPESAFE_DOWNCAST returns *object* as a *type* data object. If the conversion fails, the result of the TYPESAFE_DOWNCAST macro is 0. You should perform error checking when using the TYPESAFE_DOWNCAST macro.

## Moving from Object-based containers to the BIDS library

In ObjectWindows 1.0, the *TWindowsObject* class was derived from the class *Object* from the container class library. In ObjectWindows 2.5, the templatized BIDS container class library is used in place of the *Object*-based container class library. The BIDS library provides quicker execution times and much greater code flexibility. The BIDS templatized container classes are described in Chapter 1 of the *Borland C++ Class Libraries Guide*. This change affects code that places the *TWindow* class (the ObjectWindows 2.5 equivalent of *TWindowsObject*; see page 431) in *Object*-based containers and code that calls *Object* member functions such as *IsA*, *NameOf*, and the *isXXX* member functions such as *isEmpty*, *isFull*, *isSortable*, and so on.

Code that places the *TWindow* class in *Object*-based containers should be converted to use the BIDS templatized container classes. Otherwise, to put your own *TWindow* classes into *Object*-based containers, you would have to:

- Multiply derive your class from *Object* as well as its ObjectWindows base class.

- Implement castability for your class; see the README.TXT file for information on this procedure.
- If implementing castability (which is strongly recommended), use the TYPESAFE_DOWNCAST macro to downcast the *Object*s from the container back to your *TWindow*-derived class.

## Streaming

There have been some minor changes to the stream class library. There have also been substantial changes in how streaming is implemented for ObjectWindows 2.5 classes, although existing code should continue to work correctly with only minor modifications.

### Removed insertion and extraction operators

These operators no longer exist:

```
opstream &operator <<(opstream &, TStreamableBase *);
ipstream &operator >>(ipstream &, void * &);
```

If you were calling this << operator, you can use the following call instead:

```
opstream.WriteObjectPtr((TStreamableBase *) p);
```

This >> operator was removed because it had no real functionality.

### Implementing streaming

The Borland C++ 4.5 container class library dramatically simplifies the process of setting up your classes for streaming. The process uses the macros DECLARE_STREAMABLE and IMPLEMENT_STREAMABLEX.

The DECLARE_STREAMABLE macro can be used in a class derived from *TStreamableBase* (as most of the ObjectWindows classes are). It takes three parameters:

- Streaming class modifier, such as _OWLCLASS (these macros are discussed in the *ObjectWindows Reference Guide*)
- The class name
- Version number

Note that, though you are not required to provide a value for the first parameter, you must provide a space for it. For example:

```
class TMyClass : public TStreamableBase
{
  DECLARE_STREAMABLE(, TMyClass, 1);
};
```

The version number you use is up to you. Some streaming functions emit the version number during certain operations. You *must* put the DECLARE_STREAMABLE macro in your class definition in order to use streaming functionality with your ObjectWindows classes.

After declaring your class streamable with the DECLARE_STREAMABLE macro, you need to specify the IMPLEMENT_STREAMABLE*X* macro. This macro performs a number of steps that let you stream your class, including creating an extraction operator for your class:

```
ipstream & operator >>(ipstream &, TMyClass * &);
```

For the IMPLEMENT_STREAMABLE*X* macro, you must determine *X* to figure out which macro you should use. To do this, count the number of immediate base classes for your class plus the number of virtual base classes you want to stream. This number determines which macro you use. For example, suppose the class *TMyClass* is derived from *TFrameWindow*, which inherits *TWindow* virtually. In that case, you would use the IMPLEMENT_STREAMABLE2 macro.

You also need to provide *Read* and *Write* functions for your class. For example:

```
void
MyClass::Write(opstream &)
{
  // Whatever functionality you require...
}

void*
MyClass::Read(ipstream &, unsigned long )
{
  // Whatever functionality you require...
}
```

For more information on the DECLARE_STREAMABLE and IMPLEMENT_STREAMABLE*X* macros, and on streaming classes in general, see Chapter 2 in the *Borland C++ Class Libraries Guide*.

## MDI classes

*TWindow* in ObjectWindows 1.0 contained all the necessary support required to be an MDI child. This made it easy to create MDI applications, but caused MDI support code to be included even when your application didn't use it. ObjectWindows 2.5 provides three distinct MDI classes: *TMDIFrame*, *TMDIClient*, and *TMDIChild*. Now your application includes MDI support code *only* when using MDI classes.

In ObjectWindows 1.0, a typical MDI application worked like this:

- An instance of a specialized *TMDIFrame* class served as the application's main window.
- Instances of specialized *TWindow* classes, inserted into the frame window, served as MDI child windows.

ObjectWindows 2.5 is similar:

- An instance of a *TMDIFrame* class serves as the application's main window.
- An instance of *TMDIClient* serves as the MDI client window.

- Instances of the *TMDIChild* class, inserted into the client window, serve as MDI child windows.

There are a couple of examples that use the MDI features, named MFILEAPP and MDITEST. These examples are located in the EXAMPLES\OWL\MFILEAPP and EXAMPLES\OWL\MDITEST directories of your Borland C++ installation, respectively.

## Making the frame and client

In ObjectWindows 1.0, a typical way to use *TMDIFrame* was deriving a class from *TMDIFrame*, and instantiating an instance of that class in *TApplication::InitMainWindow*. In ObjectWindows 2.5, you can simply assign a stock *TMDIFrame* to be the main window. The default TMDIClient& parameter for the *TMDIFrame* constructor creates a default *TMDIClient* object. If you need some type of specialized *TMDIClient*, you can create the *TMDIClient* and pass it to the *TMDIFrame* constructor yourself. Using a class derived from *MDIFrame* is fine for porting your code, but your new ObjectWindows 2.5 applications shouldn't need to use a specialized *TMDIFrame*.

The following code shows how MDI clients and children were typically handled in ObjectWindows 1.0:

```
class TMyMDIFrame : public TMDIFrame
{
  public:
    TMyMDIFrame(LPSTR title, LPSTR menuName);
};

void
TMyApp::InitMainWindow()
{
  SetMainWindow(new TMyMDIFrame("Main Window", "MENU_1"));
}
```

In ObjectWindows 2.5, this code would look like this:

```
void
TMyApp::InitMainWindow()
{
  SetMainWindow(new TMDIFrame("Main Window", "MENU_1"));
}
```

If you wanted to specify a custom MDI client window, you would only have to modify the code slightly:

```
class TMyMDIClient : public TMDIClient
{
  public:
    TMyMDIClient();
};

void
TMyApp::InitMainWindow()
{
```

```
  SetMainWindow(new TMDIFrame("Main Window", "MENU_1", *new TMyMDIClient));
}
```

The reason the *TMDIFrame* constructor takes a reference to a *TMDIClient* instead of a pointer is to prevent you from constructing a *TMDIFrame* with a 0 pointer to an *MDIClient*. Using a reference parameter provides greater safety because it requires you to provide an actual object.

## Making a child window

In ObjectWindows 1.0, a child window was typically created as follows:

```
void
TMyMDIFrame::MakeNewChild()
{
  PTWindow* newMDIChild = new TMyChild(this, "new child");
  GetApplication()->MakeWindow(newMDIChild);
}
```

In ObjectWindows 2.5, this function should be a member of the *TMDIClient*-based class:

```
void
TMyMDIClient::MakeNewChild()
{
  (new TMyMDIChild(*this, "new child"))->Create();
}
```

You must use *TMDIChild* or a *TMDIChild*-derived class for MDI children. Notice the `*this` passed as the first parameter to the *TMDIChild* constructor. Again, MDI children must have a *TMDIClient* as a parent, so their constructors take a reference to *TMDIClient* instead of a pointer.

## WB_MDICHILD

The WB_MDICHILD flag is no longer defined. It was used to tell if a *TWindow* class was really an MDI child, and for a *TMDIFrame* to tell which of its children were really MDI children, and which were not (for example, a toolbar would not be implemented as an MDI child). In ObjectWindows 2.5, there is a *TMDIChild* class, and its parent is always a *TMDIClient*. Because all MDI children are derived from *TMDIChild* and are children of the *TMDIClient*, and toolbars and the like are children of a *TDecoratedMDIFrame*, there is no need for this flag anymore.

## Relocated functions

The following child-handling functions of the *TMDIFrame* class have been moved to the *TMDIClient* class:

| | |
|---|---|
| ArrangeIcons | CascadeChildren |
| CloseChildren | CMArrangeIcons |
| CMCascadeChildren | CMCloseChildren |
| CMCreateChild | CMInitChild |
| CMTileChildren | CreateChild |
| InitChild | TileChildren |

Code that used or overrode these functions should be changed to reference the *TMDIClient* instance, or be moved to a descendent of the *TMDIClient* class.

The names of the menu command handlers use the ObjectWindows 2.5 style, that is, *CMInitChild* is now *CmInitChild*.

### Replacing ActiveChild with GetActiveChild

In ObjectWindows 1.0, you could find the active MDI child by using the *PTWindow* data member, ActiveChild, of the *TMDIFrame* object. In ObjectWindows 2.5, you should use the *GetActiveChild* member function in the *TMDIClient* class.

## MainWindow variable

You should no longer set the variable *TApplication::MainWindow*. Instead you should use the *SetMainWindow* function. *SetMainWindow* takes one parameter, a *TFrameWindow* *, and returns a pointer to the old main window. If this is a new application, that is, one that has not set up a main window yet, the return value is 0.

Suppose your existing code looks something like this:

```
void
InitMainWindow()
{
  MainWindow = new TFrameWindow(0, "This window", new TWindow);
  MainWindow->AssignMenu("COMMANDS");
}
```

In ObjectWindows 1.0, this was a fairly common way of setting up your main window at the beginning of your application's execution. In ObjectWindows 2.5, class data members are either protected or private, preventing you from directly setting the value of the data members. The previous code would look something like this:

```
void
InitMainWindow()
{
  SetMainWindow(new TFrameWindow(0, "This window", new TWindow));
  MainWindow->AssignMenu("COMMANDS");
}
```

## Using a dialog as the main window

Because the *SetMainWindow* function expects a *TFrameWindow* * as a parameter, it is no longer possible to directly pass a *TDialog* or *TDialog*-derived object as the main window. To use a *TDialog* object as the main window, make a dialog window a client in a *TFrameWindow*. Then pass that *TFrameWindow* as the parameter to *SetMainWindow*. The CALC example, in the EXAMPLES\OWL\CALC directory of your Borland C++ installation, illustrates how to use a *TDialog*-derived class as a client window in a *TFrameWindow* object.

For example, suppose you had constructed a class derived from *TDialog* called *TMyDialog*, and wanted to use it as the main window. The code would look something like this:

```
SetMainWindow(new TFrameWindow(0, "My MainWindow", new TMyDialog, true));
```

There are a number of other changes you need to make if you're using a dialog as your main window:

- Destroying your dialog object does not destroy the frame. You must destroy the frame explicitly.
- You can no longer dynamically add resources directly to the dialog, because it isn't the main window. You must add the resources to the frame window. For example, suppose you added an icon to your dialog using the *SetIcon* function. You now must use the *SetIcon* function for your frame window.
- You can't just specify the caption for your dialog in the resource itself anymore. Instead you must set the caption through the frame window.
- You must set the style of the dialog box as follows:
  - Visible (WS_VISIBLE)
  - Child window (WS_CHILD)
  - It shouldn't have Minimize and Maximize buttons, drag bars, system menus, or any of the other standard frame window attributes

See page 102 for more information.

## TApplication message processing functions

The *ProcessDlgMsg*, *ProcessAccels*, and *ProcessMDIAccels* functions have been removed from the *TApplication* class. Message processing is now done by calling *TApplication*'s virtual *ProcessAppMsg* function, which calls the virtual *TWindow::PreProcessMsg* function of the window receiving the message (and up the chain of parents) until someone preprocesses the message, or until there are no more parents. At that point, it checks the applications accelerator table, and finally, if the message has not been handled, dispatches it to the window. This change greatly simplifies and automates the message processing procedure.

You might have ObjectWindows 1.0 code in which *ProcessAppMsg* is overridden to change the order in which it called the other processing functions. For example, the ObjectWindows 1.0 CALC example did this. This code isn't likely to be necessary in ObjectWindows 2.5; if you need to, however, you can override *PreProcessMsg* of the *TWindow* object or one of its parent windows.

You might also have ObjectWindows 1.0 code that extends *ProcessAccels* to process across multiple accelerator tables for different windows. This is best modified by assigning an accelerator table to each window, so that the window processes it automatically. You can also have each *TWindow* or *TWindow*-derived class override its *PreProcessMsg* function to handle its own accelerator table.

### GetModule function

The *GetModule* function has been removed from the *TWindowsObject* class. In most cases, you can simply replace a call to *GetModule* with a call to get *GetApplication*. For example, suppose you have the following code in your ObjectWindows 1.0 application:

```
GetModule()->ExecDialog(new TDialog(this, "DIALOG_1"));
```

You can convert this to ObjectWindows 2.5 by changing *GetModule* to *GetApplication*:

```
GetApplication()->ExecDialog(new TDialog(this, "DIALOG_1"));
```

Although ObjectWindows 2.5 provides *ExecDialog* for compatibility reasons, the recommended method of doing this would be to use the *Execute* command directly from the instantiated class. So the code above would become:

```
TDialog(this, "DIALOG_1").Execute();
```

This change is discussed in more detail on page 452.

The exception to this is when the *TWindow* descendent doesn't have a *TApplication* or *TApplication*-derived object defined for it (such as a DLL that isn't being used by an ObjectWindows application) and you need to use a member function of *TModule*. In this case, use the module object you construct for the DLL in your *LibMain* function. For example:

```
// Declare a global TModule pointer.
TModule *dllModule;

int FAR PASCAL
LibMain( ... )
{
  ⋮
  // Assign a value to dllModule here.
  dllModule = new TModule("My module", instance, cmdLine);
  ⋮
}

void
MyFunc()
{
  TWindow *parentAlias;

  // Use the GetParentObject function with dllModule.
  parentAlias = dllModule->GetParentObject(HWnd);
}
```

See DLLHELLO.CPP, located in the EXAMPLES\OWL\MISC directory of your Borland C++ installation, for a detailed example.

## DefXXXProc functions

The *TWindowsObject* member functions *DefCommandProc*, *DefChildProc*, *DefNotificationProc*, and *DefWndProc* have been removed from *TWindow* (the ObjectWindows 2.5 equivalent of *TWindowsObject*; see page 431) and effectively

replaced with the single function *DefaultProcessing*. This greatly simplifies message processing. To invoke default processing, just call your base class version of the event handler you are overriding, or call *DefaultProcessing*.

### Overriding

In general, it's best to handle one command or child ID notification per function. But sometimes it can be useful to handle multiple messages with one function. If you were overriding *DefCommandProc* or *DefChildProc* for this purpose, there are two main ways to port this code:

- Override the *EvCommand* function and do the message handling there. The CALC example, in the EXAMPLES\OWL\CALC directory of your Borland C++ installation, illustrates how to do this. This isn't technically default processing because *EvCommand* is called before a event handler is looked for.
- Override *DefWindowProc* and catch the commands there. *DefWindowProc* is called if an event handler was not found.

  Code that overrides *DefWndProc* also overrides *DefWindowProc*. Code that overrides *DefNotificationProc* must be ported to handle each notification at the child with a separate member function, using the EV_NOTIFY_AT_CHILD macro.

### Using DefWndProc for registered messages

If you were overriding *DefWndProc* to handle registered Windows messages (messages returned by *RegisterWindowMessage*), you don't need to do that in ObjectWindows 2.5. See the description of the EV_REGISTERED macro on page 43.

## Paint function

The declaration for the *TWindow* member function *Paint* has changed from:

```
virtual void Paint(HDC, PAINTSTRUCT _FAR &);
```

to:

```
virtual void Paint(TDC&, bool, TRect &);
```

TDC is part of the ObjectWindows 2.5 GDI encapsulation of the Windows API. You can use the TDC parameter in the same way that you used HDC. There is an **operator** *HDC()* defined for the TDC class that converts a TDC to an HDC. The **bool** and TRect& correspond directly to the *fErase* and *rcPaint* members of the PAINTSTRUCT type. The data members are initialized in the *TWindow::EvPaint* function, which is called by the default processing functions when a WM_PAINT message is received. The *EvPaint* function in turn calls the *Paint* function.

For example, suppose your ObjectWindows 1.0 code contained the following function declaration:

```
void
TWindow::Paint(HDC hdc, PAINTSTRUCT& ps)
{
```

```
    // Much code here...
}
```

You would change this in ObjectWindows 2.5 like this:

```
void
Paint(TDC& tdc, bool erase, TRect& rect)
{
  // Much code here...
}
```

## CloseWindow, ShutDownWindow, and Destroy functions

The declarations for these *TWindow* member functions have changed. The versions of these functions that took no parameters have been modified to an **int**. However, these functions also provide a default value for the **int** parameter, so your existing code should compile and run without modification.

## ForEach and FirstThat functions

The *ForEach* and *FirstThat* functions are used to iterate through the children of a window object. To use them, you pass a pointer to an iterator function as the first parameter of the *ForEach* and *FirstThat* function. This iterator function can be a normal function or a class member function. In ObjectWindows 1.0, the *ForEach* and *FirstThat* functions passed the iterator functions a **void** * for their first parameter. The iterator functions then had to cast this **void** * to a *TWindow* *. Although this works if the correct parameter type is passed, it doesn't provide for type checking. In ObjectWindows 2.5, these functions take a *TWindow* * directly:

Sample iterator functions for the *ForEach* function:

| ObjectWindows 1.0 | ObjectWindows 2.5 |
|---|---|
| void MyIterator(void *, void *) | void MyIterator(TWindow *, void *) |
| void TMyClass::MyIterator(void *, void *) | void TMyClass::MyIterator(TWindow *, void *) |

Sample iterator functions for the *FirstThat* function:

| ObjectWindows 1.0 | ObjectWindows 2.5 |
|---|---|
| bool MyIterator(void *, void *) | void MyIterator(TWindow *, void *) |
| bool TMyClass::MyIterator(void *, void *) | void TMyClass::MyIterator(TWindow *, void *) |

The functions are still used in the same way:

```
void
TMyWindow::SomeMyFunc()
{
  ForEach(MyIterator, 0);
  ForEach(&TMyClass::MyIterator, 0);
}
```

## TComboBoxData and TListBoxData classes

In ObjectWindows 1.0, the *TListBoxData* class, the transfer structure for *TListBox*, had the following two data members:

```
PArray Strings;
PArray SelStrings;
```

These members were pointers to *Object*-based *Arrays*, and held instances of the *Object*-based *String* class. These instances were the strings in the list box and the selected strings (mostly used for multi-select listboxes). Because of the move from the *Object*-based class library to the template-based BIDS libraries, and the introduction of a *string* class by the ANSI committee, the implementation of these data members has been changed for ObjectWindows 2.5 to the following:

```
TStringArray *Strings;
TStringArray *SelStrings;
```

*TStringArray* uses a BIDS array class to hold an array of *string* objects.

A similar change exists with *TComboBoxData*: the Strings data member is a *TStringArray* pointer instead of an *Array* pointer.

Though the new ANSI *string* class provides many new operators and functions, it doesn't provide a **const char** * operator like the *Object*-based class did. It instead has a *c_str* member function that must be used to get the data out of the class. This requires modifications to code that relied on the **const char** * operator of the *Object*-based *String* class. You must also use a *TStringArray* where you were previously using an *Object*-based *Array* class to get data out of a *TListBoxData* structure.

For example, using ObjectWindows 1.0, suppose you have just done a transfer and are getting a **const char** * to the first selected string. Assume *DialogTransfer* is a pointer to the transfer buffer and *ListBoxData* is a pointer to a *TListBoxData* inside of it.

```
Array& selStrings = *(DialogTransfer->ListBoxData->SelStrings);
const char *sel = (const char *)(String &)selStrings[0];
```

In ObjectWindows 2.5 this becomes:

```
TStringArray& selStrings = *(DialogTransfer->ListBoxData->SelStrings);
const char *sel = selStrings[0].c_str();
```

## TEditWindow and TFileWindow classes

The ObjectWindows 1.0 *TEditWindow* and *TFileWindow* classes have been removed from ObjectWindows and functionally replaced by *TEditSearch* and *TEditFile*, which are derived from the *TEdit* control class. The *TEditSearch* and *TEditFile* classes aren't full frame windows with menus like the previous classes, but instead are used to add editor functionality to *TFrameWindow* or *TMDIChild* windows.

There are two methods you can use to replace instances of *TFileWindow* or *TEditWindow* in your code: using the *TFileWindow* and *TEditWindow* classes defined in the OLDFILEW example program or adding *TEditSearch* and *TEditFile* classes as client windows in *TFrameWindow* or *TMDIChild* windows.

## Using the OLDFILEW example

The *TEditWindow* and *TFileWindow* classes have been implemented in the example programs EDITWND and FILEWND. You can find these examples in the EXAMPLES\OWL\OLDFILEW directory of your Borland C++ installation. The *TEditWindow* and *TFileWindow* classes defined in these examples can be used in much the same way as the original ObjectWindows 1.0 *TEditWindow* and *TFileWindow* classes. To add these classes to your programs, copy the source to your source directory for your application. If you're using just the *TEditWindow* class, you only need the files EDITWND.CPP and EDITWND.H. Because the *TFileWindow* class is based on the *TEditWindow* class, you also need the files FILEWND.CPP and FILEWND.H if you're using the *TFileWindow* class. Your source files that reference these classes need to include the appropriate header files.

Although this method works for converting your code, it's recommended that you write new code using the ObjectWindows 2.5 method of using *TEditSearch* and *TEditFile* client windows in *TFrameWindow* or *TMDIChild* windows.

## Adding TEditSearch and TEditFile client windows

You can attain the functionality of the *TEditWindow* and *TFileWindow* classes by instantiating a *TFrameWindow* or *TMDIChild* and specifying a *TEditFile* or *TEditSearch* object as a client window. Both the *TFrameWindow* and *TMDIChild* classes have a constructor that takes a *TWindow* pointer as its third parameter. It then uses the *TWindow* or *TWindow*-derived object as a client window. To specify a *TEditFile* or *TEditSearch* object as a client to one of these classes, construct the *TEditFile* or *TEditSearch* object and pass a pointer to the object to the constructor.

The following lines of code are from the FILEAPP example, located in the EXAMPLESOWLFILEAPP directory of your Borland C++ installation. They illustrate how to open a *TEditFile* client window in a *TFrameWindow* window.

```
void
TFileApp::InitMainWindow()
{
  ⋮
  SetMainWindow(new TFrameWindow(0, Name, new TEditFile));
  ⋮
}
```

The following lines of code are from the MFILEAPP example, located in the EXAMPLESOWLMFILEAPP directory of your Borland C++ installation. They illustrate how to open a *TEditFile* client window in a *TFrameWindow* window.

```
void
TMDIFileApp::CmFileNew()
{
  ⋮
  TMDIChild child(*Client, "", new TEditFile(0, 0, 0));
  ⋮
}
```

## TSearchDialog and TFileDialog classes

The *TSearchDialog* and *TFileDialog* classes have been removed from ObjectWindows. Use the *TReplaceDialog* or *TFindDialog* class in place of *TSearchDialog* and the *TFileOpenDialog* class in place of the *TFileDialog* class. These new classes are based on the class *TCommonDialog*, which encapsulates the base functionality of the Windows common dialogs.

## ActivationResponse function

The *ActivationResponse* function has been removed from the *TWindow* and *TWindowsObject* classes. Determining when a window has been activated can be done by catching the appropriate message, like WM_MDIACTIVATE, WM_ACTIVATE, or WM_SETFOCUS as appropriate. You can find an example of using WM_ACTIVATE to determine when a window is active in the SCRNSAVE example, which is located in the EXAMPLES\OWL\SCRNSAVE directory of your Borland C++ installation. You can find an example of using WM_SETFOCUS in the BSCRLAPP example, which is located in the EXAMPLES\OWL\BSCRLAPP directory of your Borland C++ installation.

## Dispatch-handling functions

The *BeforeDispatchHandler* and *AfterDispatchHandler* functions have been removed from ObjectWindows. You can obtain similar functionality by overriding *WindowProc* for a *TWindow*-derived class. The procedure for doing this is:

1. Overload the *WindowProc* function in your derived class.
2. In your *WindowProc* function, do some processing before calling the default *TBaseClass::WindowProc*.
3. Call *TBaseClass::WindowProc*.
4. Save the return value from *TBaseClass::WindowProc*.
5. Do some processing after *TBaseClass::WindowProc* has executed.
6. Return the saved return value when you exit your *WindowProc*.

For example:

```
LRESULT
TMyWindow::WindowProc(uint msg, WPARAM wParam, LPARAM lParam)
{
  // Do whatever 'before' processing you want here.
  BeforeHandling();

  LRESULT ret = TFrameWindow::WindowProc(message, wParam, lParam);

  // Do whatever 'after' processing you want here.
  AfterHandling();

  return ret;
}
```

# DispatchAMessage function

*DispatchAMessage* has been removed from ObjectWindows. Messages should be sent to the Windows API with the ObjectWindows 2.5 *SendMessage* encapsulation.

## General messages

For sending general window messages (anything other than messages that are part of WM_COMMAND, such as WM_FIRST + *XXX* messages), code would be converted as follows:

```
// Before
DispatchAMessage(WM_MESSAGE, ATMessage, &TWindow::DefWndProc)
// After
SendMessage(WM_MESSAGE, ATMessage.WParam, ATMessage.LParam);

// Before
SomeOtherWindow->DispatchAMessage(WM_FIRST + WM_MESSAGE, ATMessage, &TWindow::DefWndProc);
// After
SomeOtherWindow->SendMessage(WM_MESSAGE, ATMessage.WParam, ATMessage.LParam);
```

## The DefProc parameter

*DispatchAMessage* took a pointer to a function as its last parameter. *DispatchAMessage* called this function if a DDVT entry was not found for the message. When an ObjectWindows 2.5 window receives a message and doesn't find a handler for it, it automatically invokes the proper default handling. See page 443 for more information on default message handling.

## Command messages

There are a number of different kinds of command messages you might need to convert. Menu command messages of the form CM_FIRST + XXX are converted as follows:

```
// Before
OtherWin->DispatchAMessage(CM_FIRST + CM_MENUID,ATMessage, &TWindow::DefCommandProc);
// After
OtherWin->SendMessage(WM_COMMAND, CM_MENUID, ATMessage.LParam);
```

In ObjectWindows 2.5, command messages sent this way go directly to the specified window, *not* to the focus window.

Child ID notifications of the form ID_FIRST + XXX are converted as follows:

```
// Before
OtherWin->DispatchAMessage(ID_FIRST + ID_CHILDID,ATMessage, &TWindow::DefChildProc);
// After
OtherWin->SendMessage(WM_COMMAND, ID_CHILID, ATMessage.LParam);
```

# KBHandlerWnd

The *KBHandlerWnd* data member has been removed from the *TApplication* class. Keyboard handling is implemented through the virtual *TWindow* member function *PreProcessMsg*.

## MAXPATH

In ObjectWindows 1.0, MAXPATH was defined in the header file filewnd.h. In ObjectWindows 2.5, it no longer is. MAXPATH is defined in the header file dir.h, so if you use the MAXPATH define you should now include the standard header file dir.h.

## Style conventions

ObjectWindows 2.5 uses somewhat different style conventions from ObjectWindows 1.0. Although your application should compile fine without these stylistic changes, you should make these changes anyway to ensure easy compatibility with your future ObjectWindows code.

### Changing WinMain to OwlMain

In ObjectWindows 1.0, you used the *WinMain* function to create an instance of a *TApplication* class and call its *Run* member function. In ObjectWindows 2.5, you do this in the function *OwlMain*. ObjectWindows 2.5 provides a default *WinMain* that performs error handling and exception handling. The default *WinMain* function calls the *OwlMain* function. If you were doing any initialization in *WinMain*, you should move it to *OwlMain* and remove your *WinMain* function.

*OwlMain* differs from *WinMain* in its signature. Whereas *WinMain* takes a number of Windows-specific arguments, *OwlMain* takes an **int** and a **char** ** and returns an **int**—just like the *main* function in a traditional C or C++ program.

You still need to derive your own application class from *TApplication* to override *InitMainWindow* and *InitInstance*. *TApplication*'s constructor no longer requires you to specify the instance handles, command line, and main window show flag; the hidden *WinMain* function provides those values (you can optionally specify the name).

Here's an example of using the *OwlMain* function:

```
class TMyApp: public TApplication
{
  public:
    TMyApp(char far *name): TApplication(name) {}
    void InitMainWindow();
};

void
TMyApp::InitMainWindow()
{
  ⋮
}

int
OwlMain(int argc, char* argv[])
{
  return TMyApp("Wow!").Run();
}
```

## Data types and names

ObjectWindows 2.5 functions use Windows-style names, such as LPSTR, PWORD, and HANDLE, only when there is a direct connection between that member and something in the Windows API. An example is the connection between a event-handling function and the Windows message it handles. ObjectWindows 2.5 also avoids using Windows-style types such as PTWindowsObject and RTMessage wherever possible, and instead uses C++ type names, such as **char far ***, **unsigned short ***, and **const void ***. This helps to abstract the ObjectWindows conventions from the Windows API, and ease porting problems to other platforms in the future.

Also, function parameters in ObjectWindows 1.0 were usually named A*Something*; that is, the name was prefixed with a capital A, the first letter of the name was capitalized, and the rest of the name was in lowercase. ObjectWindows 2.5 uses a lowercase name without the capital-A prefix.

For example, the ObjectWindows 1.0 *TWindow* constructor looked like this:

```
TWindow(PTWindowsObject AParent, LPSTR ATitle, ... );
```

The ObjectWindows 2.5 *TFrameWindow* constructor (the equivalent of the ObjectWindows 1.0 *TWindow* constructor; see page 431) looks like this:

```
TFrameWindow(TWindow *parent, const char *title, ... );
```

Notice that the types *PTWindowsObject* and LPSTR have been changed to *TWindow* * and **const char ***, and the parameter names *AParent* and *ATitle* have been changed to *parent* and *title*.

OWLCVT performs these conversions for you. But unless you're careful, this can cause problems, because the conversion affects only the first instance of a variable declared on a line. For example, suppose you have the following declaration:

```
PTEdit ptEdit1, ptEdit2, ptEdit3, ptEdit4;
```

After conversion, this line would look like this:

```
TEdit _FAR * ptEdit1, ptEdit2, ptEdit3, ptEdit4;
```

Thus, instead of being pointers to *TEdit* controls, *ptEdit2*, *ptEdit3*, and *ptEdit4* are actual *TEdit* instances. You can correct this problem by changing the line so that each instance of the pointer type occurs on a separate line:

```
PTEdit ptEdit1;
PTEdit ptEdit2;
PTEdit ptEdit3;
PTEdit ptEdit4;
```

Alternatively, you can correct the line after OWLCVT has run, adding the * operator to each variable name:

```
TEdit _FAR * ptEdit1, * ptEdit2, * ptEdit3, * ptEdit4;
```

## Replacing MakeWindow with Create

ObjectWindows 2.5 uses the *TWindow::Create* function to create a window instead of the *TModule::MakeWindow* function used in ObjectWindows 1.0. Although the *Create* function existed in ObjectWindows 1.0, *MakeWindow* provided a safer way to create a

window, because it performed a certain amount of error checking before calling *Create* that calling *Create* alone did not. But ObjectWindows 2.5 makes use of C++ exceptions to catch such errors without using the explicit error-handling code that *MakeWindow* contains. You are not *required* to use *Create* in place of *MakeWindow*; *MakeWindow* still exists and can be used as before without changing code, but it is considered obsolete, and will probably be removed from future versions of the ObjectWindows class library.

### Replacing ExecDialog with Execute

ObjectWindows 2.5 uses the *TDialog::Execute* function instead of the *TModule::ExecDialog* function commonly used in ObjectWindows 1.0, for the same reasons given for using *Create* instead of *MakeWindow* in the previous section. As with *TModule::MakeWindow*, *TModule::ExecDialog* still exists and can be used as before, but is considered obsolete, and will probably be removed from future versions of the ObjectWindows class library. For example:

```
(new TDialog(MainWindow, "DIALOG_1"))->Execute();
```

### Getting the application and module instance

The application and module instance has been encapsulated in the ObjectWindows 2.5 library manager. This allows the easy manipulation of Borland- and user-defined DLLs. To facilitate this change, you should replace calls to the *GetApplication()->hInstance* function with a call to *GetLibInstance*. For example, suppose you have the following code:

```
Cursor = LoadCursor(GetApplication()->hInstance, "ThisCursor");
```

You can convert this like this:

```
Cursor = LoadCursor(GetLibInstance(IDL_APPLICATION), "ThisCursor");
```

### Defining WIN30, WIN31, and STRICT

You do not need to define WIN30, WIN31, or STRICT as long as you include owldefs.h (or a file that includes owldefs.h, such as owl.h) before you include windows.h. The owldefs.h header file defines STRICT and includes windows.h for you. But if you include windows.h before including owldefs.h, you need to define STRICT. Also, you can only target Windows 3.1 or above with ObjectWindows 2.5.

# Troubleshooting

This section lists a number of common problems you might encounter while converting your code from ObjectWindows 1.0 to ObjectWindows 2.5.

## OWLCVT errors

This section describes some common warning and error messages you might encounter when running OWLCVT on your ObjectWindows 1.0 code. Some of these messages are displayed onscreen as OWLCVT processes your code, and others are placed as comments in your converted files.

- **Unrecognized DDVT value**
  OWLCVT doesn't have a specific translation for some DDVT value. In this case, it inserts a generic value that you can search for and replace manually.

- **Cannot create backup file**
  OWLCVT creates backup copies of all the source and header files that it modifies and places them in the directory OWLBACK. When you get this warning, OWLCVT could not create the backup files for some reason.

- **Redeclaration of *var***
  This is equivalent to a compiler error telling you that you have redeclared the data item *var*.

## Compiler warnings

Here are some common warnings you might encounter when running your converted ObjectWindows 1.0 code through the Borland C++ 4.5 compiler:

- *Paint* hides function
- *ShutDownWindow* hides function
- *CloseWindow* hides function
- *DestroyWindow* hides function
- *IdleAction* hides function

For each of these functions, you might get a warning similar to this:

```
Paint(HDC, PAINTSTRUCT &) hides virtual Paint(void *, void *)
```

This can be ignored: the (**void** *, **void** *) functions were part of the Borland mechanism for providing compatibility between Windows 3.0 and 3.1. These functions were never used.

## Compiler errors

Here are some common errors you might encounter when running your converted ObjectWindows 1.0 code through the Borland C++ 4.5 compiler:

- **Type LPSTR or type X must be a struct or class name :: GetClassName**
  OWLCVT converts calls to the Windows API by preceding the call with a :: operator. If you use the name of an API function in some context other than calling a Windows API function, like overriding the *GetClassName* member function of *TWindow*, OWLCVT might add a :: operator there as well (though there are some cases it knows to ignore). This might cause the compiler to generate an error. You can fix this error by removing the :: operator that was added by OWLCVT.

- **Cannot convert 'TWindow *' to 'T*Class* *'**
  This is caused because *TWindow* is used in ObjectWindows 2.5 as a virtual base. You cannot directly downcast a *TWindow* or *TWindow* pointer to a class that is virtually derived from *TWindow*. To fix this error, use the TYPESAFE_DOWNCAST macro. For more information, see page 435.

- **Cannot cast from 'Base *' to 'Derived *'**
  Use the TYPESAFE_DOWNCAST macro to cast the Base pointer to a Derived pointer. This is essentially the same error as the previous one. For more information, see page 435.
- **Cannot convert 'int *' to 'TScrollerBase *'**
  You need to include the scroller.h header file. In ObjectWindows 1.0, this was done by owl.h, but the header file directories and layout have changed for ObjectWindows 2.5. This is discussed on page 433.

## Run-time errors

Here are some common errors you might encounter when running an application compiled from converted ObjectWindows 1.0 code:

- **Paint not getting called.**
  The declaration for the *Paint* function has changed. You need to change your *Paint* function to match the *TWindow* member function *Paint*. See page 444.
- **BeforeDispatchHandler, AfterDispatchHandler not being called.**
  See page 448.
- **FirstThat or ForEach not working.**
  It is important to stay typesafe when using multiple inheritence and virtual base class, as ObjectWindows 2.5 does. When multiple and virtual inheritence are used, the address of contained objects is not always the same as that of the objects they are inside. For example, in ObjectWindows 1.0, suppose you have a pointer to a *TDialog*, and you want to get a pointer to its base class, *TWindowsObject*. The following code would work in ObjectWindows 1.0, although it isn't typesafe because the conversion was done through a **void** pointer:

```
TDialog  *dialog_pointer;
void *void_pointer;
WindowsObject *winObj_pointer;
  ⋮
void_pointer = (void *) dialog_pointer;
winObj_pointer = (TWindowsObject*) void_pointer;
```

  In ObjectWindows 2.5, this wouldn't work. You would have to make the conversion type safe:

```
TWindow* window_pointer = (TWindow *) dialog_pointer;
```

  When the compiler knows it is converting a *TDialog* pointer to point to a virtual base, it adjusts the value of the pointer appropriately. This kind of unsafe typecasting might exist in ObjectWindows 1.0 code without breaking the code. Here is an example of this, in which *IsChild* determines if a **void** * passed in is currently a child window by using *FirstThat*:

```
bool
TMyWindow::IsChild(void * child)
{
  if (FirstThat(Test, child))
```

```
    return true;
  else return false;
}
```

where the *Test* function is:

```
bool
Test(void * winChild, void * child)
{
  return winChild == child;
}
```

Assuming *IsChild* was called with a pointer to a *TDialog* object, this code wouldn't compile correctly. After changing to passing a *TWindow* *, things work fine. When you convert this to ObjectWindows 2.5, the *Test* function takes a *TWindow* *, not a **void** *. This fails because when *IsChild* was called with a pointer to a *TDialog*, it was converted to a **void** *. The test function then compares this to *TWindow* * in a unsafe way. But the function won't work because when it was called, it was passed a *TDialog* *. Even though the *TDialog* was a child, its pointer value didn't match any of the *TWindow* pointers in the child list.

- **MDI application does not have any menu items enabled.**
  Make sure that you use the ObjectWindows 2.5 mdi.rh include file. This file contains the constants for standard items in the MDI menu, such as CM_CASCADECHILDREN. In particular, don't use the definitions from the ObjectWindows 1.0 owlrc.h include file.

# Index

## Symbols


! localization prefix 398
    automation tables 389
!= operator 227
# localization prefix 399
&= operator 228
+= operator (TRegion) 227
- (hyphen), command-line options 349
/ (slash), command-line options 349
-= operator (TRegion) 228
= operator 227
== operator 227
@ localization prefix 398
    automation tables 389
    registration tables 373
^= operator (TRegion) 229
|= operator 228


## Numerics


3-D controls, dialog boxes as 28


## A


About command (common) 87
Above member function
    TEdgeConstraint 72
Absolute member function
    TEdgeConstraint 73
abstract classes 7
accessing
    button gadget information 181
    data 129-131, 134
    document and view properties 141
    document manager 132
    gadget appearance 173
    gadget windows' font 186
    internal TDib structures 234
    TBitmap 222
    TBrush 216
    TCursor 232
    TFont 218
    TIcon 230
    TPalette 219
    TPen 215
    TRegion 226
    VBX controls 251
activating (objects)
    defined 295
    user interface for 271
activation rectangle 271
ActivationResponse member function
    TWindowsObject 448
ActiveChild member function
    TMDIFrame 441
adding
    *See also* creating
    behavior to MDI client windows 83
    event response table declarations 426
    event response table definitions 426
    event response table entries 427
    menus to views 135
    TEditFile client windows 447
    TEditSearch client windows 447
AddItem member function
    TVbxControl 254
AddRef method (IUnknown) 300
AddString member function
    TListBox 147
AddUserFormatName member function
    TOleFrame 309
AddWindow member function
    TFrameWindow 77
After enum 184
AfterDispatchHandler member function
    TWindowsObject 448
aggregation 288
    automation and 366
    defined 295
    factory callbacks 364, 365
aliases, frame windows 77
amEmbedding flag 371
amxxxx constants 364, 379
AngleArc member function
    TDC 212
AnimatePalette member function
    TPalette 220, 221
ANSI C++ standard
    changes to 418
API 204
AppDictionary variable 343
AppExpert 294
applicat.h 16
application classes 13
application dictionary
    containers
        Doc/View 306
        ObjectWindows 316
    naming variable 316
    servers
        Doc/View 342
        ObjectWindows 353
application instance, getting 452
application objects 15
    accessing 17
    attaching document manager 125
    closing 26-27
    constructing 20
    containers
        Doc/View 304
        ObjectWindows 316
    CreateOleObject method 317, 357
    creating 16, 17, 18-19
    default, overriding 18
    deriving from TOcModule 304
    handling events 137
    initializing 18, 21-24
        first instances 21
    naming 16, 18
    processing incoming messages 25
    servers
        Doc/View 343
        ObjectWindows 357
    untitled 18
application registration structures 308
application running mode 364, 379
    testing 379
applications 27
    *See also* application objects
    multiple-document interface *See* MDI applications
    single-document interface *See* SDI applications
    text-based
        find-and-replace operations 113
appname key 308, 384, 400

Arc member function
TDC 212
Arg*x* variables 388
AS_MANY_AS_NEEDED macro 193
assignment statements
menu resources 89
proxy classes 411, 416
AssignMenu member function
TFrameWindow 78, 89, 435
associating, window objects with interface elements 65, 68
AttachStream member function
TDocument 130
Attr data member
TFrameWindow 89, 435
TWindow 67
Attr structure 67
attributes
dialog boxes 103
window creation 66-68
overriding 67
AUTOARGS macro 411, 412
AutoCalc example 294, 382
automation declaration 386
automation definition 388
collections 401, 404
enum values 392
localizing
command names 398
registration 400
multiple objects, coordination of 395
proxy classes for 408
read-only property 401
registration tables 382
WinMain 385
XLAT resources 400
AUTOCALL_PROP_REF macro 415
AUTOCALL_*xxxx* macros
table of 411
autocreating dialog boxes 100-101
AUTODATA macros 386, 389
autodefs.h 408
AUTODETACH macro 396
AUTOENUM macro 392
AUTOFLAG macro 389
AUTOFUNC macros 386, 389
AutoGen utility 292, 408
source code 295
AUTOINVOKE macro 388
AUTOITERATOR macro 403
automacr.h 408
automatable members 388
declaring 385
automated application *See* automation server
automated object 295
*See also* automation server
automatic MDI child windows 83
-Automation option 349
automation
command objects 291
defined 267, 296
ObjectComponents implementation 291
RTTI required 391
supporting in applications 265
automation classes
defined in autodefs.h 408
table of 280
automation commands
arguments, exposing 390
data type specifiers 390
Help context IDs for 390
hooks for 387
localizing names 397
naming 390
validating arguments 387
automation controllers 407-416
CallCalc example 294
compiling and linking 412-413
creating allocator 408
declaring proxies 408-409, 415
defined 296, 407
enumerating collections 413-416
header files 408
initializing OLE libraries 408
localizing 397, 409
passing
objects by value 415
parameters 411-412
sending commands 407, 415
steps for creating 407
automation declaration tables 385-388
*See also* automation tables
AutoCalc example 386
automation definition tables, vs. 385, 389
macros for 386
no terminating macro 386
automation definition tables 388-393
*See also* automation tables
AutoCalc example 388
automation declaration tables, vs. 385, 389
data type specifiers 390-392
delegation in 395
marking localized strings 398
automation iterator *See* iterator
automation macros
automacr.h, defined in 408
declaration tables
AUTODATA 386
AUTODETACH 396
AUTOFUNC 386
AUTOITERATOR 403
AUTOPROP 386
DECLARE_AUTOCLASS 385
definition tables 389-390
AUTOENUM 392
DEFINE_AUTOCLASS 388
DEFINE_AUTOENUM 392
END_AUTOCLASS 386, 388
END_AUTOENUM 392
EXPOSE_APPLICATION 390
EXPOSE_DELEGATE 395
EXPOSE_INHERIT 395
EXPOSE_ITERATOR 403
EXPOSE_METHOD 411
EXPOSE_*xxxx* 389
OPTIONAL_ARG 411
REQUIRED_ARG 389
hooks 388
proxy classes
AUTOARGS 411
AUTOCALL_PROP_REF 415
AUTOCALL_*xxxx* 411
AUTONAMES 411
automation objects
coordinating in one server 395
invalidating 396
-Automation option 383
automation proxy classes *See* proxy classes
automation servers
aggregating 366
AutoCalc example 294, 382
collections, automating 400-405
command arguments, exposing 390
compiling and linking 394
coordinating multiple automated objects 395

data type specifiers 390
declaring automatable members 385
defined 296
defining automatable members 388
delegating to C++ objects 395
DLL servers, as 374
documentation strings 390
enum values, automating 392
factory templates 385
header files 394
invalidating automated objects 396
localizing command names 397
naming commands 390
read-only properties (example) 401
receiving commands 407
registrar objects 385
registration 382
registration keys, table of 383
single-use recommended 383
steps for creating 381
type libraries 405
WinMain 385
automation tables 291
*See also* automation declaration tables, automation definition tables
localizing 389
AUTONAMES macro 411
AUTONAMES macros 411
AUTONAMES0 macro 414
AUTONOHOOK macro 388
AUTOPROP macros 386, 389
AUTORECORD macro 388
AUTOREPORT macro 388
autosubclassing 28
AUTOUNDO macro 388
AUTOVALIDATE macro 388


# B


backing up your old source files 420
base classes, function calls 129
Before enum 184
BeforeDispatchHandler member function
TWindowsObject 448
BEGIN_REGISTRATION macro 122, 310, 372
BeginDocument member function
TPrintout 200
BeginPath member function
TDC 211
BeginPrinting member function
TPrintout 200
Below member function
TEdgeConstraint 72
BI_APP_DLL macro 376
Bind member function
TAutoProxy 412
BitBlt member function
TDC 212
BITMAP structure
convert TBitmap class to 222
BITMAPINFO operator
TDib 234
BITMAPINFOHEADER operator, TDib 234
BitsPixel member function
TBitmap 223
BOCOLE support library 275, 288
custom interfaces 288
defined 296
source code for 288
BOCOLE.DLL 275
Borland C++ OLE support *See* ObjectComponents Framework
Borland Custom Controls Library 27
Bottom enum 80
bounding a gadget 173
brush origin
getting 209
setting 209
building MDI applications 82-84
buttons, dialog boxes, processing 102, 114
BWCC.DLL 27
BWCC32.DLL 27
BWCCEnabled member function
TApplication 27


# C


C++ applications
converting to OLE servers 359-371
C++ containers
documents, creating 333
examples 329
handlers
view events 335
WM_OCEVENT 334
header files 339
memory allocator for 331
registrar object 330
registration 329
steps for creating 328
view windows 332
WinMain 330
C++ servers
client windows 366
compiling and linking 371
document lists 361
documents
creating 367
painting 369
examples 360
factory callbacks 363
header files 371
initializing OLE 360
main window, managing 370
memory allocator 360
message loop 363
multiple document types 368
registrar object 362
registration 360
steps for creating 359
view windows 366
WinMain 362
WM_OCEVENT handlers 368, 370
CallCalc example 294, 415
Cancel button, processing 102
CanClose member function
TApplication 26-27
TDocument 132
TWindow 102
CapsLock enum 191
captions, main windows 21
capturing mouse movements 186
castability 437
CATCH macro 64
catching exceptions 102
change notifications
from Doc/View servers 350
ChangeModeToPal member function, TDib 235
ChangeModeToRGB member function, TDib 235
changing
applications, closing behavior 26
display modes, main windows 25
document templates 125
encapsulated GDI functions 204
frame windows 78
header files 433
menu item text 55
menu objects 86
Checked enum 52, 57

CheckValid member function
  TDC 213
child ID notification macros 46
child ID-based messages 428
  responding to 427
child interface elements 34
child windows
  attributes 67
  dialog boxes vs. 101
  layout metrics 70
  lists 34
  MDI applications 81, 83
    creating 83-84
ChildList interface object data
  member 34
Chord member function
  TDC 212
class hierarchies 5, 9
classes
  *See also* specific Borland C++
    class
  abstract 7
  application 13
  control 11
  derived 15, 18, 105
  deriving new 5
  dialog box 10
  document 117, 120, 128-134
    processing events 132
  document template 121-122
  exceptions 59, 62-63
  graphics 12
  instantiating 6
  MDI 438
  module 13
  nested 107
  printing 13
  shared 258
  string 446
  template 120
    constructing 123
    instantiation 122, 123-124
  types of member
    functions 8-9
  VBX control 246
  view 117, 120, 134-136
  window 10
ClearFlag member function
  TDocTemplate 125
client windows
  *See also* view windows
  containers 319
  creating in SDI
    applications 320
  dialog boxes and 98
  in OWLOCF2 example 356
  MDI applications 83
  required for
    ObjectComponents 319
  servers 356, 366
clients (OLE) *See* automation
  controllers, containers
Clip data member
  TGadget 174
clip rectangle functions 210
clip region functions 210
Clipboard formats
  registering 309
Clone member function
  TXBase 60
  TXOwl 62
cloning exception objects 60, 62
Close member function
  TDocument 130
  TOcDocument 334, 368
CloseFigure member function
  TDC 211
CloseWindow member function
  TDialog 100, 102
  TWindow 445
closing
  application objects 15, 26-27
  dialog boxes 100, 102
  documents 131-132
  view objects 136
clsid key 308, 309, 384, 412
  for multiple documents 383
CM_CREATECHILD 83
CM_EDITCONVERT 312
CM_EDITINSERTOBJECT 312,
  327
CM_EDITLINKS 312
CM_EDITOBJECT 312, 327
CM_EDITPASTELINK 312
CM_EDITPASTESPECIAL 312
CmCancel member function
  TDialog 100, 102
CmCreateChild member
  function
  TMDIClient 83
cmdline key 383, 384
CmOk member function
  TDialog 100, 102
collections
  iterating in servers
    *See also* iterators
collection classes
  AutoCalc example 404
  creating for automation 401
collection iterator *See* iterator
collection objects
  defined 400
  members, optional 404
collections
  automating 400-405
  enumerating in controllers
    enumerator object 414
    steps for enumerating 413
  examples
    AutoCalc server 401
    CallCalc controller 415
color common dialog boxes 109
Color data member
  TData 109, 110
colors 109
  fonts 110
  replacing
    gadget colors with system
      colors 174, 179, 182
    standard interface colors
      with system colors 237
COM object 296
  *See also* Component Object
    Model
combining automated C++
  objects 395
combo boxes 162
command enabling 49-58
  *See also* menu commands
  checking receiver 53
  enabler interface 51
  messages 49-51
  toggles 52, 56
command messages 449
  macros 43
  responding to 427
CommandEnable member
  function
  TButtonGadget 182
  TGadget 175
  TGadgetWindow 187
command-enabling
  objects 51-53
  changing text 55
  constructing 52
  disabling 52, 53-55
  setting text 52, 55
command-line options (OLE)
  combining 349
  DLL servers and 350
  initial character 349
  processing in servers 349
  registering 383
  table of 349
commands
  automation *See* automation
    commands
  menu *See* menu commands

Commit member function
  TDocument 131
  TOleDocument 312
CommitTransactedStorage member function
  TOleDocument 313
common dialog boxes 10, 97, 107-115
  color 109
  constructing 107-108
  error handling 108
  executing 108
  file open 111
  file save 112
  find and replace 113-114
  flags 108
  font 110-111
  modality 108
  printer 115
compilation
  automation controllers 412
  containers
    C++ 340
    Doc/View 313
    ObjectWindows 328
  servers
    automation 394
    C++ 371
    DLL 377
    Doc/View 352
    ObjectWindows 359
compiler errors 453
compiler warnings 453
Component Object Model (COM), defined 296
compound documents
  defined 296
  Doc/View container, in 312
compound files 312
  defined 296
  dtAutoOpen flag needed 313
  transacted mode 313
configuration files
  conversion and 419, 421
connector objects 288
  automation 291
  communication with 289
  defined 296
  interaction with user classes 290
constraints, layout windows 70-74
constructing
  command-enabling objects 52
  decorated frame window objects 79
  device context objects 207
  document manager 126-127
  document objects 129
  exceptions 60
  frame window aliases 77
  frame window objects 76
  menu objects 85
  template classes 123
  view objects 134
  virtual bases 435
  window objects 65-66
constructors
  TApplication 16, 18
  TBitmap 221
  TBrush 215
  TCommandEnabler 52
  TControlBar 189
  TCursor 231
  TDC 207
  TDialog 98
  TDib 233
  TDocManager 126
  TDocument 129
  TFont 217
  TFrameWindow 66, 76, 77
  TGadget 171
  TGadgetWindow 183
  TIcon 229
  TMenu 85
  TMenuDescr 93
  TMessageBar 190
  TPalette 218
  TPen 213
  TPrinter 196
  TRegion 224
  TStatusBar 191
  TToolBox 192
  TVbxControl 247
  TView 134
  TWindow 66
  TXBase 60
  TXOwl 62
ContainerGroup enum 92
containers
  *See also* C++ containers, Doc/View containers, OWL containers
  creating 264
  defined 297
  Edit menu (illustration) 268, 274
  examples
    CPPOCF1 329
    MdiOle 295
    OWLOCF1 316
    SdiOle 295
  moving from Object-based to BIDS library 436
  part objects in 299
  registration keys 308
  registration tables 307
  servers and 347, 356
  user interface 305
  verbs 272
Contains member function
  TRegion 226
control bars 189
control classes 11
  derived 105
  names 143
control elements, associating control objects with 105
control objects 143
  associating with control elements 105
control values 164
controllers *See* automation controllers
controls 11, 143, 147
  *See also* resources
  as gadgets 182
  buttons 151
    constructing 151
    responding to 152
  changing attributes 146
  check boxes 152
    constructing 153
    modifying 153
    querying 154
  class names 143
  combo boxes 162, 164
    choosing 163
    constructing 163
    querying 164
    varieties of 162
  communicating with 147
  constructing 144
  constructor parameters 145
  default style 146, 148, 151
  destroying 146
  dialog boxes 103-106
    setting up 106
  edit controls 113, 159
    Clipboard 160
    constructing 159
    edit menu 160
    modifying 161
    querying 160
  gauges 158, 159
  group boxes 154
    constructing 154
    responding to 155
  grouping 154
  initializing 144, 146
  instance variables 165
    defining 165

- interface objects and 104-106
- list boxes 148
  - constructing 148
  - modifying 148
  - querying 149
  - responding to 149
- manipulating 147
- messages 104
- pointer to 145
- radio buttons 152
  - constructing 153
- scroll bars 155
  - constructing 155
  - controlling 155
  - modifying 156
  - querying 156
  - range 155
  - responding to 156
  - thumb tracking 157
- showing 146
- sliders 158
- static 150
  - constructing 150
  - modifying 151
  - querying 151
- transfer buffers 164
  - combo boxes 167
  - defining 165
  - dialog boxes 168
  - list boxes 166
  - windows and 167
- transfer mechanism 165
- TransferData 169
- values, setting and reading 164

conventions, style 450

conversion
- checklist 422
- configuration files 419, 421
- makefiles 419, 421
- operators 215, 216, 218, 219, 222, 226, 230, 232, 234
- procedures 424
- response files 419, 421

Convert command 272, 312
- *See also* Edit menu

converting 418, 420, 432
- DefChildProc 443
- DefCommandProc 443
- DefNotificationProc 443
- DefWndProc 443
- from DDVTs to event response tables 424
- MDI classes 438
- Object-based containers to BIDS library 436
- ObjectWindows 1.0 code to ObjectWindows 2.0 417
- replacing
  - ActiveChild with GetActiveChild 441
  - ExecDialog with Execute 452
  - MakeWindow with Create 451
- STRICT define and 418
- TWindowsObject to TWindow 431
- window constructors 431
- WinMain to OwlMain 450

coordinate functions 210

coordinates (screen)
- pop-up menus 88

Copies data member
- TData 115

CPPOCF1 example 329
- creating documents 333
- OC_VIEWPARTINVALID event 335
- painting parts 335
- registration table 330
- view windows, code for 332
- WinMain 330

CPPOCF2 example 360
- documents, creating 368
- event handlers
  - OC_APPSHUTDOWN 371
  - OC_VIEWCLOSE 369
- factory callback 364
- main window, hiding 363
- message loop 363
- registration tables 361
- view windows, managing 366
- WinMain 362

Create interface object function 31

Create member function
- TBitmap 224
- TDialog 99, 101
- TGadgetWindow 183
- TPalette 220
- TWindow 68, 101, 183, 451

CreateAnyDoc member function
- TDocManager 126

CreateAnyView member function, TDocManager 126

CreateOcView member function
- TOleWindow 321, 357
  - creating
    - TOcRemView 357

CreateOleObject member function
- application object 357
- document list, and 358
- multiple document types 358

creating
- *See also* adding; constructing
- application objects 16, 17, 18-19
- child interface elements 35
- common dialog boxes 107-108
- dialog boxes 98
  - as client window 98
  - as main window 102-103
  - automatically 100-101
  - multiple 100
- document classes 128
- document registration tables 122-123
- document templates 120
- interface objects 31
- MDI child windows 83-84
- MDI frame windows 82
- menu descriptors 92, 93
- printer objects 195-196
- template class instances 123-124
- view classes 134
- window interface elements 68

CTL3D32.DLL 27

Ctl3dEnabled member function
- TApplication 28

CTL3DV2.DLL 27

current documents, viewing 120

current position functions 210

CustColors data member
- TData 109

custom control libraries 27-28

custom controls 143

CustomFilter data member
- TData 111


# D


data 117
- accessing 129-131, 134
- clearing changes 131
- protecting 131
- retrieving 101
- saving changes 131
- viewing 132, 134, 135

data items, registering 122

data members, inheritance 7

data type specifiers
- derived from TAutoVal 391
- table of 391

data validation 239-244

databases 113, 130
DC data member
  TData 110
DDVTs, converting 424
-Debug option 349
debugdesc key 376, 384, 400
debugger key 376, 384
debugging
  DLL servers 377-378, 379
  registration keys for 376
debugprogid key 376, 384
  -Debug switch and 349
declarations
  *See also* automation declarations
  document registration tables 123
  proxies 408-409, 415
  proxy collection classes 413
  response tables 40
DECLARE_AUTOCLASS macro 385
DECLARE_RESPONSE_TABLE macro 40
DECLARE_STREAMABLE macro 437
declaring
  automatable members 385
decorated frame windows 78-80
  *See also* frame windows
  constructing 79
  decorating 80
decorated windows 10
decorations 11
default directories 111
default placeholder functions 9
default printers 196
DefChildProc function
  converting 443
DefCommandProc function
  converting 443
DefExt data member
  TData 111
DEFINE_APP_DICTIONARY macro 306, 343, 353
DEFINE_AUTOCLASS macro 388
DEFINE_AUTOENUM macro 392
DEFINE_DOC_TEMPLATE _CLASS macro 121
DEFINE_RESPONSE_TABLE macro 40
defining automatable members 388
defining regions in device contexts 224
definitions
  automation *See* automation definition tables
  OLE terms 295
DefNotificationProc function
  converting 443
DefWndProc function
  converting 443
delegation
  automated C++ objects, in 395
deleting
  *See also* destroying
  dialog boxes 99, 100, 101
    automatically 100
  interface objects 33
  parent windows 100
derived classes 5
  applications 15, 18
  controls 105
  TXBase 61
  TXOwl 62
deriving from
  TGadgetWindow 187
description key 308, 309, 358, 384, 400
  localization recommended 384
designing document template classes 121-122
Destroy member function
  TDialog 100, 102
  TWindow 445
destroying
  *See also* deleting
  device context objects 207
  exceptions 60, 62
  interface elements 33
  interface objects 33
  windows 36
destructors
  TDib 233
  TDocument 131
  TGadget 172
  TGadgetWindow 183
  TMessageBar 190
  TRegion 224
  TXBase 60
  TXOwl 62
DetachStream member function
  TDocument 130
device contexts 206
  brush origin 209
  classes 12
  color functions 209
  constructing 207
  coordinate functions 210
  current position functions 210
  destroying 207
  drawing attribute functions 209
  font functions 211
  functions 208
  metafile functions 210
  object data members and functions 213
  operators 207
  output functions 211
  palette functions 209
  path functions 211
  resetting 208
  restoring 208
  retrieving information about 208
  saving 208
  TClientDC class 206
  TCreatedDC class 206
  TDC class 206
  TDesktopDC class 206
  TDibDC class 206
  TIC class 206
  TMemoryDC class 206
  TMetaFileDC class 206
  TPaintDC class 206
  TPrintDC class 206
  TScreenDC class 206
  TWindowDC class 206
  viewport mapping functions 210
  window mapping functions 210
dialog box classes 10
dialog box objects 97
dialog boxes
  as the main window 441
  attributes 103
  child windows vs. 101
  closing 100, 102
  common *See* common dialog boxes
  constructing 98
    as client window 98
    as main window 102-103
    automatically 100-101
    multiple 100
  controls 103-106
  Convert 272
  customizing 97, 100
  deleting 99, 100, 101
    automatically 100
  displaying 99, 101
  encapsulating 97, 106

error handling 101
executing 98-101
Insert Object 268, 273
modal 98, 107, 108
closing 102
modeless 99, 101, 108, 113
parent windows and 98, 100
resource IDs 98, 104
responding to messages 104, 113
retrieving data 101
simple 107
subclassing as 3-D controls 28
temporary 97
verifying input 102
DIB information 235
dictionary *See* application dictionary
direct mode 313
directories, setting default 111
directory key 308
DisableAutoCreate member function, TWindow 100
disabling
command-enabling objects 52, 53-55
exceptions 64
disabling a gadget 174
dispatch IDs 291, 414
DispatchAMessage member function, TWindowsObject 449
DISPID_NEWENUM constant 414
display modes
main windows 24-25
displaying dialog boxes 99, 101
DLL servers 374-380
client compatibility 375
command-line options and 350
compiling and linking 377
debugging 377-378, 379
defined 297
EXE servers vs. 374, 376
idle processing in 374
incompatibility between 16-bit and 32-bit 374
loading and running 379
MDI interface disallowed 375
registering 293, 375, 379
registration tables 376
running as executables 379
code for 295
steps for creating 375
tools for programming 379
DllEntryPoint function 256, 257
DllRun utility 292, 295, 379
DLLs 255, 260
32-bit entry function 256
application dictionaries and 342, 353
classes and 258
DllEntryPoint 256, 257
encapsulating 15
entry and exit functions 256
_export keyword 257
export macros 258
exporting functions 257
function calls and 255
_import keyword 258
import macros 258
importing functions 257
LibMain 256
loading 261
mixing with static libraries 261
multiple processes and 342, 353
non- 260
opening predefined 27-28
_OWLDLL macro 260
shared classes and 258
start-up code 256
static data and 255
TModule and 258, 259
WEP 256
writing 255
dmMDI constant 126
dmMenu constant 126
dmNoRevert constant 126
dmSaveEnabled constant 126
dmSDI constant 126
Doc/View applications
converting to OLE servers 341-352
Doc/View containers
application dictionary 306
compiling 314
compound documents, managing 312
Edit menu support 311
example 304
header files 314
OwlMain 310
registrar object 310
registration 306
steps for creating 303
Doc/View model 117, 120-121, 136-142
handling events 136-139
predefined macros 138
Doc/View objects 117
setting properties 139-142
Doc/View servers
compiling and linking 352
factory callback 348
header files 351
loading and saving objects 351
OLE classes 344
OwlMain 348
registrar object 347
registration 344
sending change notifications 350
steps for creating 341
docfilter key 308, 384
docflags key 308, 384
document classes 117, 120, 128-134
accessing data 129-131
implementing 129, 132
processing events 132
protecting data 131
document lists 358
servers
C++ 361
ObjectWindows 355
document manager 117, 120-121
accessing 132
attaching to documents 125-126
constructing 126-127
Doc/View instances and 121
handling events 127-128
MDI applications 126, 127
OLE applications 126
SDI applications 126, 127
document objects 119
attaching document manager 125-126
clearing data changes 131
closing 131-132
committing data 131
constructing 129
flags, document type 123
opening 117
setting properties 139-142
viewing current 120
viewing data 132, 134, 135
document objects (ObjectComponents)
deleting 334
document registration structures 308
TRegList 355
document registration tables
CPPOCF2 example 361
passing to document template 347

document string 390
document templates 117, 121-125
  changing 125
  class instances, creating 123-124
  creating 120
  designing classes 121-122
  in non-Doc/View applications 355
  modifying 125
  registration tables 122-123, 347
  servers and 355, 358, 361
documentation (printing conventions) 3
documents 119
  creating
    C++ containers 333
    C++ servers 367
    ObjectWindows servers 356
  defined 297
docview.h 138
DoubleShadow enum 181
downcasting virtual bases to derived types 435
DPtoLP member function
  TDC 210
Draw member function
  TOcPart 336
DrawFocusRect member function
  TDC 212
DrawIcon member function
  TDC 212
drawing
  attribute functions 209
  tool functions 209
DrawMenuBar member function
  TWindow 86
DrawText member function
  TDC 212
dtAutoDelete constant 123
dtAutoOpen constant 123
dtAutoOpen flag 313
dtHidden constant 123, 384
dtNoAutoView constant 123
dtSingleView constant 123
dynamic-link libraries *See* DLLs

## E

each-instant initialization 21, 23
edit controls 159, 242
  as dialog boxes 113
  linking to validators 242
Edit Links command 312
Edit menu (OLE)
  commands 268
  Doc/View containers 311
  Object command 272
  ObjectWindows containers 327
  verbs 272, 274
Edit verb 272
EditGroup enum 92
editing (OLE)
  *See also* activating
  in-place 269
  open 274
Ellipse member function
  TDC 212
embedded objects
  defined 297
-Embedding option 349
embedding
  *See also* linking and embedding
  user interface for 268, 271
-Embedding option 358, 363
  amEmbedding flag, and 371
Enable member function
  TCommandEnabler 52, 54
EnableAutoCreate member function
  TWindow 100
EnableBWCC member function
  TApplication 27
EnableCtl3d member function
  TApplication 28
EnableCtl3dAutosubclass member function
  TApplication 28
enabling a gadget 174
enabling command-enabling objects 51, 53-55
encapsulation
  dialog boxes 97, 106
  DLLs 15
  GDI functions 204
  Windows applications 15
END_AUTOCLASS macro 386, 388
END_AUTOENUM macro 392
END_REGISTRATION macro 372
END_RESPONSE_TABLE macro 40
EndDocument member function
  TPrintout 200
EndPath member function
  TDC 211
EndPrinting member function
  TPrintout 200
enumerating
  automated collections 402, 413
  automated values 392
  data types specifiers for 393
EnumFontFamilies member function
  TDC 211
EnumFonts member function
  TDC 211
EnumMetaFile member function
  TDC 210
Error member function
  TData 108
  TValidator 244
errors 452
  common dialog boxes 108
  compiler 453
  dialog boxes 101
  run-time 454
EV_COMMAND macro 43
EV_COMMAND_AND_ID macro 43
EV_COMMAND_ENABLE macro 43, 50
EV_LBN_DBLCLK macro 428
EV_MESSAGE macro 44
EV_OWLNOTIFY macro 132
EV_REGISTERED macro 44
EV_VBXEVENTNAME macro 246, 249
EvActivateApp member function
  TOleFrame 322
EvAppBorderSpaceSet
  TOleFrame method 328
EvAppBorderSpaceSet member function
  TOleFrame 312
EvCommand member function
  TOleWindow 323
EvCommandEnable member function
  TOleWindow 323
EvDropFiles member function
  TOleWindow 323
event handlers, overriding in OLE programs 322
event response functions 114
event response tables 424
eventhan.h 39
events 39-48
  application objects 137
  Doc/View models 136-139

document manager 127-128
handling 424
notification macros 132
view objects 135, 136, 138
customizing 138-139
events handlers 39
EvLButtonDblClk member function
TOleWindow 322
EvLButtonDown member function
TOleWindow 322
EvLButtonUp member function
TOleWindow 322
EvMdiActivate member function
TOleWindow 322
EvMouseActivate member function
TOleWindow 322
EvMouseMove member function
TOleWindow 322
EvPaint member function
TOleWindow 323
EvRButtonDown member function
TOleWindow 322
EvSetCursor member function
TOleWindow 322
EvSetFocus member function
TOleWindow 322
EvSize member function
TLayoutWindow 75
TOleWindow 322
EvTimer member function
TOleFrame 322
EvVbxDispatch member function
TVbxEventHandler 249
example programs
*See also* AppExpert
automation controllers
CallCalc 415
automation servers
AutoCalc 382
ObjectComponents 294
examples
containers
Doc/View 304
MdiOle 295
localization
AutoCalc 398
Localize 294
servers
DLL 375
ObjectWindows 375
exception classes 59, 62-63
exception objects 60-61
cloning 60, 62
exceptions 59-64
catching TXWindow 102
constructing 60
destroying 60
disabling 64
macros 61, 63-64
message strings 62
throwing 61
ExcludeClipRect member function
TDC 210
ExcludeUpdateRgn member function
TDC 210
EXE servers 374
defined 297
DLL servers vs. 374
ExecDialog member function
TDialog 452
Execute interface object function 31
Execute member function
TDialog 98, 101, 109, 452
executing
common dialog boxes 108
dialog boxes 98-101
_export keyword 257
export macros 258
exporting functions 257
EXPOSE_APPLICATION macro 390
EXPOSE_DELEGATE macro 395
EXPOSE_INHERIT macro 395
EXPOSE_ITERATOR macro 403
EXPOSE_METHOD macro 389, 411
EXPOSE_PROP macros 389
EXPOSE_*xxxx* macros 389
extending TBitmap 224
extending TPalette 220
ExtendSelection enum 191
extension key 308, 384
ExtFloodFill member function
TDC 212
extra message processing 25
extraction operators 437
ExtTextOut member function
TDC 212

# F

factories 300
automation 383
factory callbacks 290, 331
actions performed 364
example code 364
prototype 364
servers
C++ 363
Doc/View 348
user-defined 364
factory templates
automation servers 385
calling CreateOleObject 357
linking and embedding servers
C++ 364
Doc/View 348
ObjectWindows without Doc/View 356
TOcAutoFactory 385
TOleDocViewFactory 348
TOleFactory 356
File menu (common)
document manager 120, 126
file names
default, specifying 111
selecting 112
file open common dialog boxes 111
file save common dialog boxes 112
FileGroup enum 92
FileName data member
TData 111
files
filtering 111, 125
linking and embedding from 273
opening 111
saving 112
FillPath member function
TDC 211
FillRect member function
TDC 212
FillRgn member function
TDC 212
Filter data member
TData 111
FilterIndex data member
TData 111
filters 111, 125
find and replace common dialog boxes 113-114
Find Next button, processing 114
Find Next command (common) 114

find-and-replace
operations 113-114
finding next occurrence 114
handling messages 113
FindColor member function
TDib 236
FindIndex member function
TDib 236
FINDMSGSTRING message 113
FindProperty member function
TDocument 141
TView 141
FirstGadget member function
TGadgetWindow 187
first-instance initialization 21
FirstThat interface object
function 34, 37
FirstThat member function
TWindowsObject 445
flags
common dialog boxes 108, 114
Doc/View properties 141
document type 123
find-and-replace
operations 114
fonts 110
frame windows 77
menus 88
WB_MDICHILD 440
Flags data member
TData 113
Flags member function
TData 108
FlattenPath member function
TDC 211
FloodFill member function
TDC 212
FlushDoc member function
TDocManager 132
font common dialog
boxes 110-111
font functions 211
fonts 110
FontType data member
TData 110
ForEach interface object
function 34, 37
ForEach member function
TWindowsObject 445
format*n* key 308, 400
FR_DIALOGTERM flag 114
FR_FINDNEXT flag 114
frame windows 10, 75-78
*See also* decorated frame
windows
aliases 77
C++ servers 366, 370
changing 78
closing 26
constructing 76-78
creating 23
hiding server's 363
MDI applications 82
menu objects 89-95
shrinking 76
specifying client window 76, 80
FrameRect member function
TDC 212
FrameRgn member function
TDC 212
free-floating menus 88
FromPage data member
TData 115
function calls
base classes 129
functions
*See also* member functions
ActivationResponse 448
AfterDispatchHandler 448
BeforeDispatchHandler 448
CloseWindow 445
Create 451
Destroy 445
DispatchAMessage 449
event response 114
ExecDialog 452
Execute 452
FirstThat 445
font 211
ForEach 445
GetModule 443
MakeWindow 451
metafile 210
output 211
Paint 444
path 211
ShutDownWindow 445
TApplication message
processing 442
update 210
update region 210
VBXInit 245
VBXTerm 245
window mapping 210

# G

gadget windows 182
accessing font 186
capturing mouse movements
for gadgets 186
classes 189
constructing 183
control bars 189
converting 188
creating 183
deriving from
TGadgetWindow 187
desired size 188
displaying mode
indicators 191
idle action processing 187
inner rectangle 188
inserting gadgets 183
laying out gadgets 184
layout units
border 188
layout 188
pixels 188
message bars 190
message response
functions 189
notifying when gadgets
change size 185
objects 171
painting 187
positioning gadgets 185
removing gadgets 184
searching through
gadgets 187
setting
hint mode 186
layout direction 184
window margins 184
shrink wrapping 185
status bars 191
tiling gadgets 185
tool boxes 192
GadgetChangedSize member
function
TGadgetWindow 185
GadgetFromPoint member
function
TGadgetWindow 187
GadgetReleaseCapture member
function
TGadgetWindow 186
gadgets 11
accessing
appearance 173
button gadget
information 181

associating
events 172
strings 172
border style 173
border width 173
bounding rectangle 173
button state 180
capturing mouse
movements 186
classes 177
cleaning up 175
clipping rectangle 174
command buttons 180
command enabling 182
controls as gadgets 182
corner notching 181
creating button gadgets 180
deriving from TGadget 175
disabling 174
displaying
bitmaps 179
text 178
enabling 174
expand to fit available
room 174
identifiers
event 172
gadget 172
string 172
identifying 172
initializing 175
inserting
into gadget windows 183
into status bars 191
invalidating 176
laying out in gadget
windows 184
margin width 173
matching colors to system
colors 174, 179, 182
modifying appearance 173
mouse events 176
painting 175
in gadget windows 188
pressing button gadgets 180
removing from gadget
windows 184
searching in gadget
windows 187
separating gadgets in a
window 178
setting
border style 173
border widths 173
button gadget style 181
buttons 180
margins 173
shrink wrapping 173
sizing 174
TGadget base class 171
tiling 185
updating 176
GadgetSetCapture member
function
TGadget 177
TGadgetWindow 186
GadgetWithId member function
TGadgetWindow 187
GDI objects 13
base class 203
device contexts 206
restoring 208
selecting 208
selecting stock objects 209
support classes 13
general messages 449
responding to 428
generating
automation proxy classes 408
generic messages 44
get DC logical coordinates as
absolute physical
coordinates 208
GetActiveChild member
function
TMDIFrame 441
GetAppDescriptor global
function 396
GetApplication member function
TDocManager 443
TWindow 17, 443
GetAspectRatioFilter member
function
TDC 211
GetAttributeHDC member
function
TDC 213
GetBitmapBits member function
TBitmap 223
GetBitmapDimension member
function
TBitmap 223
GetBits member function
TDib 234
GetBkColor member function
TDC 210
GetBkMode member function
TDC 210
GetBorders member function
TGadget 173
GetBorderStyle member function
TGadget 173
GetBounds member function
TGadget 173
GetBoundsRect member
function, TDC 210
GetButtonState member function
TButtonGadget 181
GetButtonType member function
TButtonGadget 181
GetCharABCWidths member
function, TDC 211
GetCharWidth member function
TDC 211
GetChildLayoutMetrics member
function, TLayoutWindow 75
GetClipBox member function
TDC 210
GetClipRgn member function
TDC 210
GetColor member function
TDib 236
GetColors member function
TDib 234
GetCurrentPosition member
function, TDC 211, 212
GetDCOrg member function
TDC 208
GetDefaultExt member function
TDocTemplate 125
GetDescription member function
TDocTemplate 125
GetDesiredSize member function
TGadget 174
TGadgetWindow 188
GetDeviceCaps member
function, TDC 208
GetDialogInfo member function
TPrintout 199
GetDIBits member function
TDC 212
GetDirection member function
TGadgetWindow 184
GetDirectory member function
TDocTemplate 125
GetDocManager member
function, TDocument 126, 132
GetEnabled member function
TGadget 175
GetEventIndex member function
TVbxControl 251
GetEventName member
function, TVbxControl 251
GetFileFilter member function
TDocTemplate 125
GetFlags member function
TDocTemplate 125
GetFont member function
TGadgetWindow 186

GetFontData member function
TDC 211
GetFontHeight member function
TGadgetWindow 186
GetHandled member function
TCommandEnabler 53
GetHDC member function
TDC 213
GetHintMode member function
TGadgetWindow 186
GetIconInfo member function
TCursor 232
TIcon 231
GetId member function
TGadget 172
GetIndex member function
TDib 236
GetIndices member function
TDib 234
GetInfo member function
TDib 234
GetInfoHeader member function
TDib 234
GetInnerRect member function
TGadget 176
TGadgetWindow 188
GetMapMode member function
TDC 210
GetMargins member function
TGadget 173
TGadgetWindow 188
GetMenuDescr member function
TFrameWindow 78
GetModule member function
TWindowsObject 443
GetNearestColor member
function, TDC 209
GetNearestPaletteIndex member
function, TPalette 219
GetNumEntries member
function, TPalette 220
GetNumEvents member
function, TVbxControl 251
GetNumProps member function
TVbxControl 252
GetObject member function
TBitmap 222, 223
TBrush 216
TFont 218
TPalette 219
TPen 215
GetPaletteEntries member
function, TPalette 220
GetPaletteEntry member
function, TPalette 220
GetPixel member function
TDC 212
GetPolyFillMode member
function, TDC 210
GetProcAddress function 258
GetProp member function
TVbxControl 252
GetPropIndex member function
TVbxControl 252
GetPropName member function
TVbxControl 252
GetRect member function
TOcPart 335
GetRgnBox member function
TRegion 226
GetROP2 member function
TDC 210
GetStretchBltMode member
function, TDC 210
GetSystemPaletteEntries
member function, TDC 209
GetSystemPaletteUse member
function, TDC 209
GetText member function
TTextGadget 178
GetTextColor member function
TDC 210
getting
application instance 452
brush origin 209
module instance 452
GetViewName member function
TView 135
GetViewportExt member
function, TDC 210
GetViewportOrg member
function, TDC 210
GetWindow member function
TView 135
GetWindowExt member
function, TDC 210
GetWindowOrg member
function, TDC 210
global functions
GetAppDescriptor 396
MostDerived 396
globally unique identifer *See*
GUID
glossary of OLE terms 295
graphics, bitmaps 133
graphics classes 12
graphics objects
base class 203
grapples 271
defined 301
GrayString member function
TDC 212
grouping pop-up menus 93
GUID (globally unique
identifier)
defined 297
GUID (globally unique
indentifier)
generating 293, 295
GuidGen utility 293
source code 295

## H

Handle data member
TDC 213
HANDLE operator
TDib 234
HANDLE_MSG macro 334
HANDLE_OCF macro 335
Handled data member
TCommandEnabler 53
handles, window 31
handling
messages and events 424
VBX control messages 249
HasNextPage member function
TPrintout 197
HasPage member function
TPrintout 200
HBITMAP operator
TBitmap 222
HBRUSH operator
TBrush 216
HCURSOR operator
TCursor 232
HDC (TDC) device-context
operator 207
header files 433
automation controllers 408
containers
C++ 339
Doc/View 314
ObjectWindows 328
directories 433
servers
automation 394
C++ 371
Doc/View 351
ObjectWindows 359
Height member function
TBitmap 223
TDib 235
Help (online),
ObjectComponents and
OLE 294

Help files
automation commands, for 390
for type libraries 405
helpdir key 384, 406
HelpGroup enum 92
HFONT operator
TFont 218
HICON operator
TIcon 230
hiding server windows 363
high-priority idle processing 25
hInstance parameter 15
hooks
automation commands 387
Horizontal enum 183, 193
HPALETTE operator
TPalette 219
HPEN operator
TPen 215
hPrevInstance data member
TApplication 21
hPrevInstance parameter 15
HRGN operator
TRegion 226
HWindow data member
TDialog 106
HWindow interface object data member 31
HWndReceiver data member
TCommandEnabler 53


## I

IBContainer (BOCOLE interface) 288
IBDocument (BOCOLE interface) 288
ICONINFO
convert TCursor object to 232
convert TIcon object to 231
icons, conventions, documentation 3
Id data member
TCommandEnabler 53
IDCANCEL message 99, 100
identifying a gadget 172
IDispatch interface 291, 408
defined 298
idle processing (messages) 25
IdleAction
in DLL servers 374
IdleAction member function
TApplication 25
TGadgetWindow 187
IDOK message 98, 100
IDW_TOOLBAR identifier 312
IEnumVARIANT interface 291, 404, 413, 414
IMPLEMENT_STREAMABLE macro 437
implementing streaming 437
import macros 258
importing functions 257
inactive state
user interface for 271
include path
conversions 419, 421, 422, 433
indeterminate constraints (windows) 74
Indeterminate enum 52, 57
inheritance
data members 7
OLE classes 344
Init member function
TDC 213
TFrameWindow 66
TWindow 66
InitApplication member function
TApplication 18, 21
InitChild member function
TMDIClient 83
InitialDir data member
TData 111
initialization
application objects 18, 21-24
each instance 21, 23
first instances 21
main windows 21, 23-24
initializing OLE libraries 331
initilizing OLE libraries
in C++ containers 331
InitInstance member function
TApplication 18, 23, 320
InitMainWindow member function
TApplication 18, 82, 102, 320
in-place editing
defined 298
user interface for 269
in-process servers See DLL servers
input
common dialog boxes and 115
dialog boxes 99, 102
verifying 102
filtering 243
processing user 23, 25
unexpected user 59
input dialog boxes 107
input validators 239
inputdia.rc 107
Insert member function
TDecoratedFrame 80
TGadgetWindow 183
TStatusBar 191
Insert Object command 312
user interface for 273
Insert Object dialog box 268
description key, and 358
linking 273
registration database, and 269
insertable key 361
Inserted member function
TGadget 175
inserting
gadgets into gadget windows 183
gadgets into status bars 191
OLE objects 268
insertion operators 437
instance variable 165
InstanceCount data member
TXBase 60
instantiation
applications 19, 21, 23
classes 6
template classes 122, 123-124
InStream member function
TDocument 130
integrated development environment See IDE
interface elements
associating with window objects 65, 68
destroying 33
making visible 31
parent and child 34
properties 32
interface objects 29, 30
associating with controls 104-106
creating 31
deleting 33
destroying 33
document views 135-136
members
ChildList 34
Create 31
Execute 31
FirstThat 34, 37
ForEach 34, 37
HWindow 31
IsWindowVisible 32
Parent 34
SetupWindow 31
Show 32

pointers 135
properties 32
interfaces (COM) 287
automation 291
defined 298
interfaces (OLE)
BOCOLE support library 296
IntersectClipRect member function, TDC 210
intializing OLE libraries
in C++ servers 360
Invalidate member function
TGadget 176
TWindow 109
InvalidateObject member function, TAppDescriptor 396
InvalidatePart member function
TOleView 350
InvalidateRect member function
TGadget 176
invalidating automation objects 396
invalidating gadgets 176
InvertRect member function
TDC 212
InvertRgn member function
TDC 212
Invoke command object method 291
IsDirty member function
TDocument 131
IsFlagSet member function
TDocTemplate 125
TFrameWindow 77
IsOK member function
TDib 235
IsOpen member function
TDocument 131
IsOptionSet member function
TOcModule 379
TRegistrar 363
IsPM member function
TDib 235
IsReceiver member function
TCommandEnabler 53
IsValid member function
TValidator 243
IsValidInput member function
TValidator 243
IsWindowVisible interface object function 32
iterators
_NewEnum name 403
automation declaration 403
automation definitions 403
user-implemented 404
ITypeInfo interface 291
ITypeLib interface 291
IUnknown interface 288
aggregation and 295, 365
defined 298
part of COM 296

## K

KBHandlerWnd data member
TApplication 449
keys *See* registration keys
keystrokes, validating 243

## L

language key 384
-Language option 349, 350, 400
laying out gadgets 184
layout constraints (windows) 70-74
layout direction 184
Layout member function
TLayoutWindow 74
layout metrics (child windows) 70
layout units 188
converting 188
layout windows 69-75
creating 74
defining constraints 70, 73, 74
LayoutSession member function
TGadgetWindow 184
LayoutUnitsToPixels member function
TGadgetWindow 188
LButtonDown member function
TGadget 177
LButtonUp member function
TGadget 177
LCIDs *See* locale IDs
Left enum 80
LeftOf member function
TEdgeConstraint 72
LibMain function 256
parameters 15
libraries
custom control 27-28
dynamic-link *See* DLLs
OLE, initializing 360
*See also* BOCOLE support library
servers and 352, 377
LineDDA member function
TDC 212
LineTo member function
TDC 212
link source 330, 334
defined 299
linked objects
defined 298
linking (OLE objects)
user interface for 273
linking and embedding
defined 266
from files 273
how ObjectComponents implements 288
list boxes 148
lists, automating *See* collections
lmBottom enum 71
lmCenter enum 71
lmHeight enum 71
lmLayoutUnits enum 72
lmLeft enum 71
lmPixels enum 72
lmRight enum 71
lmTop enum 71
lmWidth enum 71
LoadLibrary function 258
locale IDs 350, 397
overriding system default 400
locale.h 399
locale.rh 397
localization 397-400
*See also* language key
automation controllers 409
automation servers 397
automation tables 389
defined 299
examples, AutoCalc 398, 400
language-neutral strings 398
Localize example 294
prefixes (!, @, #) 398
registration keys 400
registration tables 373
string retrieval algorithm 399
TLocaleString 399
XLAT resources 397
localization tables *See* XLAT resources
Localize example 294
LOGBRUSH structure
convert TBrush class to 216
LogFont data member
TData 110
LOGFONT structure
convert TFont class to 218
logical coordinates
get as absolute physical coordinates 208
LOGPEN structure
convert TPen class to 215

low-priority idle processing 25
LPARAM parameter 429
lpCmdLine parameter 15
LPtoDP member function
  TDC 210

# M

MacroGen utility 293
macros 260
  *See also* automation macros, message cracker macros, registration macros; specific macro
  application dictionary 306, 353
  child ID notification 46
  command message 43
  document flags 123
  document registration tables 122
  document templates 121
  events 132
  exceptions 61, 63-64
  export 258
  generating new 293
  generic messages 44
  import 258
  MAKELANGID 399
  registered messages 44
  registration, data items 122
main windows
  captions 21
  closing 26
  dialog boxes as 102-103, 441
  display modes 24-25
    changing 25
  initializing 21, 23-24
MainWindow variable 441
makefiles
  conversion and 419, 421
  ObjectComponents 286
MAKELANGID macro 399
MakeWindow member function
  TWindow 451
making the frame and client 439
manipulating
  child windows 37
  MDI child windows 83
manual MDI child window creation 84
MapColor member function
  TDib 236
MapIndex member function
  TDib 237
MapUIColors member function
  TBitmapGadget 180
  TDib 237
MaskBlt member function
  TDC 213
matching gadget colors to system colors 174, 179, 182
maximizing windows 24
MaxPage data member
  TData 115
MAXPATH macro 450
MDI applications 81-84
  building 82-84
  converting to servers 351, 359
  document manager 126, 127
MDI classes 438
MDI interface
  DLL servers disallow 375
MDI windows 10, 80-84
  child 81, 83-84
  client 83
  frame 82
MdiOle example 295
member functions 8-9
  overriding
    TDocument 129
    TView 134, 135
memory allocator
  C++ containers 331
  C++ servers 360
memory models
  automation controllers 412
  servers
    automation 394
    C++ 371
    DLL 377
    ObjectWindows 352
menu bars 90
  updating 86
menu commands
  *See also* command enabling
  responding to 113
menu descriptors 90-95
  adding separators 93
  creating 92, 93
  grouping pop-up menus 93
  merging menus 94
menu merging 94
  menu descriptors used for 325
  ObjectWindows container 324
  user interface for 269
menu objects 85
  adding menu descriptors 90, 91-92, 93
  changing 86
  constructing 85
  frame windows 89-95
  pop-up menus 88-89, 93
  retrieving information 86-87
  system 87
menu resources 89, 92, 435
  adding separators 93
  STEP10.RC example 325
menuname key 361, 400
menus 14
  assigning to a frame window 435
  free-floating 88
  merging 94
  view objects 135
MergeMenu member function
  TFrameWindow 78
merging menus *See* menu merging
message bars 190
message cracker macros
  HANDLE_MSG 334
  HANDLE_OCF 335
message cracking 45
  ObjectComponents messages 334
message loops
  C++ servers 363
  factory callbacks 364
  ObjectComponents applications, in 331
message queues
  DLL servers 374
MessageLoop member function
  TApplication 25
message-processing functions 442
messages
  *See also* Windows messages
  child ID-based 427, 428
  child windows 83
  command 43, 427, 449
  command-enabling 49-51
  control notification codes 428
  dialog boxes
    handling find-and-replace 113
    responding to 104, 113
  exceptions 62
  general 428, 449
  generic 44
  handling 424
  idle processing 25
  notification 428
  ObjectComponents events 289
  registered 44

sending to controls 104
using DefWndProc for registered 444
window layouts 75
Windows applications 25
WM_COMMAND_ENABLE 182
WM_SYSCOLORCHANGE 182
metafile functions 210
Microsoft 3-D Controls Library 27, 28
autosubclassing 28
MinPage data member
TData 115
mixing object behavior 6
modal dialog boxes 98, 107
as common 108
closing 102
modeless dialog boxes 99
as common 108, 113
creating 101
modes
application running 364
compound file I/O 313
modifying gadget appearance 173
*See also* changing
ModifyWorldTransform member function
TDC 210
module classes 13
module instance
getting 452
module.h 16
MostDerived global function 396
mouse clicks
handling in
ObjectComponents 323
mouse event handlers
overriding for OLE 322
mouse events in a gadget 176
MouseEnter member function
TGadget 177
MouseLeave member function
TGadget 177
MouseMove member function
TGadget 177
Move member function
TVbxControl 254
MoveTo member function
TDC 210, 212
moving from Object-based containers to BIDS library 436
multimedia files 130
multiple document types
servers
C++ 368
non-Doc/View 358
ObjectWindows 357
multiple printers 196
multiple-document interface applications *See* MDI applications

## N

naming applications 16, 18
nCmdShow data member
TApplication 24
nCmdShow parameter 15, 24
nCmdShow variable 363
nested classes 107
_NewEnum iterator name 403
NextGadget member function
TGadgetWindow 187
NextProperty property 140
NO_CPP_EXCEPTIONS macro 64
NoHints enum 186
non-Doc/View servers *See* ObjectWindows servers
non-ObjectWindows servers *See* C++ servers
nonvirtual functions 8
-NoRegValidate option 349
notification messages
responding to 428
servers 350
NOTIFY_SIG macro 138
notifying gadgets window 185
NotifyViews member function
TDocument 132, 133
NumColors member function
TDib 235
NumLock enum 191
NumScans member function
TDib 235

## O

Object command (OLE) 272, 312
object data members and functions 213
object handle 204
Object Linking and Embedding *See* OLE
ObjectComponents Framework applications
*See also* containers, servers, controllers
creating 264
interaction with 276
automation implementation 291
BOCOLE support library 275
capabilities 263
client windows required 319
creating .REG files with 350
defined 275-276, 299
documentation 293
example programs 294
getting started 264, 265
glossary of terms 295
handling Windows events 322
internal operation of 287
message cracking 334
messages 289
ObjectWindows Library, and 276
RTTI required 315
tools for programming 292
ObjectGroup enum 92
objects 6
*See also* specific object
command-enabling 51-53
disabling 52, 53-55
exception 60-61
exceptions 62
view 119-120
objects (COM) 296
objects (embedded) 297
objects (linked) 298
objects (OLE)
activating 271, 337
converting format 272
creating *See* factory callbacks
editing in place 269
inserting 268, 338
linking 273
loading and saving
with Doc/View 350
selecting 271
verbs 272
objects, automated *See* automation servers
ObjectWindows applications
converting to OLE servers 352-359
ObjectWindows containers
*See also* Doc/View containers
application dictionary 316
application object 316

client windows 319
compiling and linking 328
CreateOleObject, implementing 317
documents, creating 321
header files 328
menu merging 324
OLE classes for 320
OwlMain 319
registrar object 318
registration 317
steps for creating 315
tool bar 328
user interface programming 322
ObjectWindows Library
defined 299
ObjectComponents Framework, and 276
OLE classes 305
OLE support *See* ObjectComponents Framework
ObjectWindows servers
*See also* Doc/View servers
application dictionary 353
application object 357
client windows 356
compiling and linking 359
document lists 355
documents
creating 356
helper objects for 356
registering 355
examples
OWLOCF2 352
Tic Tac Toe 375
header files 359
OwlMain 355
registrar objects 355
registration 353
steps for creating 352
OC_APPSHUTDOWN 371
OC_VIEWCLOSE 369
OC_VIEWPARTINVALID 335
OCF *See* ObjectComponents Framework
ocfevx.h 368
OCFMAKE.GEN 371
ocrEmbedSource format 309
ocrLinkSource format 309
ocrMetafilePict format 309
ocrMultipleUse constant 383
OffsetClipRgn member function
TDC 210
OffsetViewportOrg member function, TDC 210
OffsetWindowOrg member function, TDC 210
ofTransacted file mode 313
OK button, processing 102
OLB file extension 301
OLE
compound files 313
defined 266, 299
dialog boxes 268
documentation 293
file modes 313
glossary of terms 295
interfaces 287
support in Borland C++ *See* ObjectComponents Framework
support library *See* BOCOLE support library
user interface *See* user interface (OLE)
OLE applications
*See also* containers, servers, controllers
document manager 126
OLE classes
containers
Doc/View 305
ObjectWindows 320
inheritance 305, 344
servers
Doc/View 344
ObjectWindows 356
OLE clients *See* automation controllers, containers
OLE libraries, initializing 331, 360
OLE objects 344
automating 415
OLE servers *See* automation servers, C++ servers, Doc/View servers, ObjectWindows servers
oleframe.h 351
olemdifr.h 351
OLETOOLS (for IDE) 293
open editing
defined 299
user interface for 274
Open member function
TDocument 130
TOleDocument 312
Open verb 272, 274
opening documents 117
opening files
common dialog boxes and 111
opening predefined DLLs 27-28
operators
!= (TRegion) 227
&= (TRegion) 228
+= (TRegion) 227
-= (TRegion) 228
= (TRegion) 227
== (TRegion) 227
^= (TRegion) 229
|= (TRegion) 228
HRGN (TRegion) 226
OPTIONAL_ARG macro 411
OrgBrush data member
TDC 213
OrgFont data member
TDC 213
OrgPalette data member
TDC 213
OrgPen data member
TDC 213
OrgTextBrush data member
TDC 213
outer IUnknown pointer *See* aggregation
output
common dialog boxes and 115
output functions 211
OutStream member function
TDocument 130
overriding
child window attributes 67
window creation attributes 67
virtual functions 129
Overtype enum 191
OWLCVT 419
_OWLDLL macro 260
OWLFastWindowFrame member function, TDC 213
OwlMain
containers
Doc/View 310
ObjectWindows 319
servers
Doc/View 348
ObjectWindows 355
OwlMain function 19-20
OWLOCF1 example container 316
registration table 318
OWLOCF2 example server 352
client windows 356
registration tables 353
OWLOCFMK.GEN 352

# P


paginating printouts 199
Paint member function
  TGadget 176
  TGadgetWindow 187
  TWindow 195, 199, 444
PaintBorder member function
  TGadget 175
PaintGadgets member function
  TGadgetWindow 188
painting
  gadget windows 187
  gadgets 175
  server documents
    C++ 369
    Doc/View 350
PaintRgn member function
  TDC 212
palette mode 235
Paradox, linking tables from 273
parameters
  LibMain function 15
parent interface elements 34
Parent interface object data member 34
parent windows
  destroying 100
  dialog boxes and 98, 100
parts 288
  defined 299
  enumerating 336
  painting 335
Paste command *See* Edit menu
Paste Link command 312
Paste Link command *See* Edit menu
Paste Special command 312
Paste Special command *See* Edit menu
PatBlt member function
  TDC 212
path functions 211
path key 384
PathToRegion member function
  TDC 211
PercentOf member function
  TEdgeConstraint 73
permid key 308, 384
permname key 308, 384, 400
physical coordinates
  get logical coordinates as 208
Pie member function
  TDC 212
Planes member function
  TBitmap 223
PlayMetaFile member function
  TDC 210
PlayMetaFileRecord member function
  TDC 210
PlgBlt member function
  TDC 213
point size, setting 110
pointers
  applications 17
  interface objects 135
PointSize data member
  TData 110
PolyBezier member function
  TDC 212
PolyBezierTo member function
  TDC 212
PolyDraw member function
  TDC 212
Polygon member function
  TDC 212
Polyline member function
  TDC 212
PolylineTo member function
  TDC 212
PolyPolygon member function
  TDC 212
PolyPolyline member function
  TDC 212
pop-up menus 88-89, 93
PositionGadget member function
  TGadgetWindow 185
predefined Doc/View event handlers 138
PreOpen member function
  TOleDocument 313
PressHints enum 186
PrevProperty property 140
print job dialog box 115
Print member function
  TPrinter 199
print setup dialog box 115
printer common dialog boxes 115
printer devices 201
printer objects 195-201
  creating 195-196
  overriding system default 196
  selecting printer devices 201
printers
  configuring 201
  default 196
    overriding 196
  multiple 196
  selecting 196
printing 115, 195
  documents 199-200
  window contents 198
printing classes 13
printing conventions (documentation) 3
printout objects
  constructing 197
  indicating further pages 200
  paginating 199
  printing 199, 200
  summary 199
  window contents 198, 199
    constructing 198
PrintPage member function
  TPrintout 195, 197, 200
processing user input 23, 25
  dialog boxes and 99, 102
processing Windows messages 25
progid key 308, 309, 384
  automation servers 383
  controller binds with 412
  localization not possible 384
program controllers *See* automation controllers
project files, conversion and 422
prompts 107
properties
  automated collections as 413, 415
  documents 139-142
  exposing automated 389
  interface elements 32
  interface objects 32
  macros for controllers 411
  read-only automated 401
  template 125
  view objects 139-142
PropertyFlags member function
  TDocument 141
  TView 141
PropertyName member function
  TDocument 141
protecting data 131
prototypes, factory callback 364
proxy classes 408-412
  *See also* automation controllers
  assignment and 411, 416
  collections, for 413
  constructors 409
  declaring 408-409
  defined 408
  derived from
    TAutoProxy 409
  example 408

generating 292
generating automatically 408
implementing 410-412
specifying arguments 411
proxy objects
creating 412
declaring 415
pseudo-GDI objects 203
PtIn member function
TGadget 176
PtVisible member function
TDC 210
pure virtual functions 9

## Q

Quattro Pro
inserting objects from 269
QueryInterface, IUnknown 288
aggregating, used in 295
QueryViews member function
TDocument 132, 133

## R

RealizePalette member function
TDC 209
recording automation commands 387
RecordingMacro enum 191
Rectangle member function
TDC 212
RectVisible member function
TDC 210
reference counting
defined 300
part of COM 296
Refresh member function
TVbxControl 254
.REG files, creating 350
REG.DAT 301, 372
REGDATA macro 122, 309, 372, 374
regDoc variable 362
REGDOCFLAGS macro 123, 309
RegEdit utility 350
viewing registration database 373
REGFORMAT macro 309
Register utility 293, 379
source code 295
registered messages
macros 44
using DefWndProc for 444
registering
*See also* unregistering
Clipboard formats 309
code for 295
DLL servers 293, 295
.REG files 350
servers 352
terminating application after 385
registrar objects 347-348, 355, 361
defined 300
containers
C++ 330
ObjectWindows 318
creating 310
naming variable 310
servers
automation 385
C++ 362
Doc/View 347
ObjectWindows 355
registration
command-line options 383
containers
C++ 329
Doc/View 306
ObjectWindows 317
defined 300
localizing keys 400
RegTest example 294
servers
automation 382
C++ 360
DLL 375
Doc/View 344
keys, table of 346
ObjectWindows 353
registration database 361, 372-374
*See also* servers
Insert Object dialog box 269
keys 345, 372
recording information 372
viewing 373
registration keys
appname 308, 384, 400
clsid 383, 412
cmdline 383
debugging keys 376
description 358
helpdir 406
insertable 361
menuname 361
progid 383, 412
serverctx 375
tables
automation servers 383
containers 308
linking and embedding servers 346
localizable 400
typehelp 390, 406
usage 383
user-defined 374
registration macros 309-310, 372-374
BEGIN_REGISTRATION 310, 372
END_REGISTRATION 372
REGDATA 309, 372, 374
REGDOCFLAGS 309
REGFORMAT 309
REGISTRATION_FORMAT_BUFFER 309, 347
REGITEM 374
registration tables
buffer expansion 347
building 310
condionalizing for DLL servers 376
containers 307
CPPOCF2 example 361
creating 372
defined 300
document 122-123
example 353
servers
automation 382
C++ 361
Doc/View 344
ObjectWindows 353
REGISTRATION_FORMAT_BUFFER macro 309, 347
REGITEM macro 374
RegLinkHead variable 355
-RegServer option 349, 350
RegTest example 294
Release member function
IUnknown interface 300
ReleaseObject member function
TOcRemView 368
TOcView 334
remote views
*See also* TOcRemView
C++ servers 367
creating 357
defined 300
releasing 368
Remove member function
TGadgetWindow 184
RemoveChildLayoutMetrics member function
TLayoutWindow 75
Removed member function
TGadget 175
RemoveItem member function
TVbxControl 254

removing gadgets from gadget windows 184
replace standard interface colors with system colors 174, 179, 182, 237
replacing text 113
REQUIRED_ARG macro 389
reset a device context 208
reset origin of a brush object 216
ResetDC member function
 TDC 208
ResizePalette member function
 TPalette 220
resource IDs 98, 104
resource scripts
 XLAT resources 397
resources 434
responding to
 child ID-based messages 427
 command messages 427
 general messages 428
 menu selections 113
 messages 104, 113
 notification messages 428
response files
 conversion and 419, 421
response tables 39
 command-enabling messages 50
 declaring 40
 defining 40, 41
 example 40
 macros 40
  child ID notification 46
  command message 43
  generic messages 44
  message cracking and 45
  registered messages 44
  Windows messages 45
 view objects 132, 138
restore a device context 208
RestoreBrush member function
 TDC 209
RestoreDC member function
 TDC 208
RestoreFont member function
 TDC 209
RestoreMenu member function
 TFrameWindow 78
RestoreObjects member function
 TDC 209
RestorePalette member function
 TDC 209
RestorePen member function
 TDC 209
RestoreTextBrush member function, TDC 209
restoring GDI objects 208
result string 269, 273
RETHROW macro 64
retrieve information about a device context 208
Revert member function
 TDocument 131
RGB mode 235
Right enum 80
RightOf member function
 TEdgeConstraint 72
RoundRect member function
 TDC 212
RTTI
 automation, type checking 391
 required for ObjectComponents 315
Run member function
 TApplication 16
 TOcRegistrar 362, 363
 TRegistrar 311, 385
running DLL servers 293
running mode, application 364, 379
 testing 379
run-time errors 454
run-time management (applications) 15

## S

SameAs member function
 TEdgeConstraint 73
SaveDC member function
 TDC 208
saving device contexts 208
saving files 112
ScaleViewportExt member function
 TDC 210
ScaleWindowExt member function
 TDC 210
scope resolution operator 432
screen coordinates
 pop-up menus 88
scroll bars 155
ScrollDC member function
 TDC 212
ScrollLock enum 191
SDI applications
 converting to servers 351, 359
 document manager 126, 127
SdiOle example 268, 295
searches
 find-and-replace operations 113-114
  finding next occurrence 114
  handling messages 113
searching through gadgets in gadget windows 187
SelectClipPath member function
 TDC 211
SelectClipRgn member function
 TDC 210
SelectEmbedded member function, TOleWindow 324
SelectImage member function
 TBitmapGadget 179
selecting objects
 defined 301
 GDI 208
 inactive 272
 stock 209
 user interface for 271
selection rectangle 272
 dashed for links 273
SelectObject member function
 TDC 208
SelectStockObject member function, TDC 209
SendDlgItemMessage member function, TDialog 104
separators (menu objects) 93
server applications *See* servers
serverctx key 375
servers
 DLL *See* DLL servers
 EXE 297
 registering 352
 verbs 272
servers (linking and embedding)
 *See also* C++ servers, Doc/View servers, ObjectWindows servers
 containers and 347, 356
 creating 265
 examples
  Step15 344
  Tic Tac Toe 295
 multiple documents and 357
 registration keys, table of 346
servers (OLE)
 *See also* automation servers, servers (linking and embedding)
 as standalone application 349
 defined 301
 DLL 374-380

Set member function
TEdgeConstraint 70, 72
SetAntialiasEdges member function, TButtonGadget 181
SetBitmapBits member function
TBitmap 223
SetBitmapDimension member function, TBitmap 223
SetBkColor member function
TDC 210
SetBkMode member function
TDC 210
SetBorders member function
TGadget 173
SetBorderStyle member function
TGadget 173
SetBounds member function
TGadget 173
SetBoundsRect member function
TDC 210
SetButtonState member function
TButtonGadget 181
SetCheck member function
TCommandEnabler 52, 57
SetChildLayoutMetrics member function, TLayoutWindow 75
SetColor member function
TDib 236
SetDefaultExt member function
TDocTemplate 125
SetDescription member function
TDocTemplate 125
SetDIBits member function
TDC 212
SetDIBitsToDevice member function, TDC 212
SetDirection member function
TGadgetWindow 184
TToolBox 193
SetDirectory member function
TDocTemplate 125
SetDocManager member function
TApplication 125
TDocument 132
SetDocTitle member function
TView 135
SetEnabled member function
TGadget 175
SetFileFilter member function
TDocTemplate 125
SetFlag member function
TDocTemplate 125
SetHintCommand member function, TGadgetWindow 186
SetHintMode member function
TGadgetWindow 186
SetHintText member function
TMessageBar 190
SetIcon member function
TFrameWindow 78
SetIndex member function
TDib 236
SetMainWindow member function, TApplication 23, 25
SetMapMode member function
TDC 210
SetMapperFlags member function, TDC 211
SetMargins member function
TGadget 173
TGadgetWindow 184
SetMenu member function
TFrameWindow 78
SetMenuDescr member function
TFrameWindow 78, 94
SetMiterLimit member function
TDC 210
SetModeIndicator member function, TStatusBar 192
SetNotchCorners member function, TButtonGadget 181
SetOuter member function
TOcApp 365
TOcRemView 365
SetPaletteEntries member function, TPalette 220
SetPaletteEntry member function
TPalette 220
SetPixel member function
TDC 212
SetPolyFillMode member function, TDC 210
SetPrinter member function
TPrinter 201
SetPrintParams member function
TPrintout 199
SetProp member function
TVbxControl 253
SetRectRgn member function
TRegion 226
SetROP2 member function
TDC 210
SetShadowStyle member function, TButtonGadget 181
SetShrinkWrap member function
TGadget 173
TGadgetWindow 185
SetSize member function
TGadget 174
SetSpacing member function
TStatusBar 192
SetStretchBltMode member function, TDC 210
SetSystemPaletteUse member function
TDC 209
SetText member function
TCommandEnabler 52, 55
TMessageBar 190
TTextGadget 179
SetTextColor member function
TDC 210
setting
brush origin 209
hint mode 186
hint text 190
layout direction 184
message bar text 190
window margins 184
Setup member function
TPrinter 201
SetupWindow interface object function 31
SetupWindow member function
TDialog 106
TOcApp 370
TOcRemView 367
TOcView 334
TOleFrame 290
TWindow 145, 146
SetValidator member function
TEdit 242
SetViewportExt member function, TDC 210
SetViewportOrg member function, TDC 210
SetWindowExt member function
TDC 210
SetWindowOrg member function, TDC 210
SetWorldTransform member function, TDC 210
shared classes 258
ShouldDelete data member
TDC 213
Show interface object function 32
ShowWindow member function
TWindow 99, 101
shrink wrapping
gadget windows 185
gadgets 173
shrinking frame windows 76
ShutDownWindow member function, TWindow 445

simple dialog boxes 107
single-document interface applications *See* SDI applications
SingleShadow enum 181
Size member function
  TDib 235
SizeMax data member
  TData 110
SizeMin data member
  TData 110
sizing a gadget 174
spacing status bar gadgets 192
SpeedMenu, verbs on 272
standard Windows controls 11
StartScan member function
  TDib 235
static controls
  constructing 151
  default style 151
status bars 191
STEP10.RC 325
stock objects 209
storages 296, 313
stream class library 437
streaming 437
  implementing 437
streams
  document classes and 130
StretchBlt member function
  TDC 212
StretchDIBits member function
  TDC 212
STRICT, defining 418, 452
string class 446
strings
  exceptions 62
  localizing *See* localization
StrokeAndFillPath member function, TDC 211
StrokePath member function
  TDC 211
structures
  VBXEVENT 249, 250
style conventions 450
Style data member
  TData 110
styles, window 32
support library *See* BOCOLE support library
switches *See* command-line options
switching to
  palette mode 235
  RGB mode 235
symbol names, localizing *See* localization
SysColorChange member function
  TBitmapGadget 180
  TButtonGadget 182
  TGadget 174
System menu, generating 87
system registration database *See* registration database
system registry *See* registration database

# T

TabbedTextOut member function, TDC 212
tables, registration
  defined 300
TAppDescriptor class 396
  members
    InvalidateObject 396
TApplication class 15
  building objects 16-18
  calling members 17, 21
  closing 26-27
  constructors 16, 18
    passing WinMain parameters 20
  creating MDI applications 82
  creating the main window 68
  getting the application instance 452
  header file 16
  initializing 18, 21-24
    first instances 21
  instantiation 19, 21, 23
  members
    BWCCEnabled 27
    CanClose 26-27
    Ctl3dEnabled 28
    EnableBWCC 27
    EnableCtl3d 28
    EnableCtl3dAutosubclass 28
    hPrevInstance 21
    IdleAction 25
    InitApplication 18, 21
    InitInstance 18, 23, 320
    InitMainWindow 18, 82, 102, 320
    KBHandlerWnd 449
    MessageLoop 25
    nCmdShow 24
    Run 16
    SetDocManager 125
    SetMainWindow 23, 25
  message processing functions 442
  overriding members 18, 23
  passing command parameters to 20
  pointers to objects 17
TAutoBase class 396
TAutoBool data type specifier 391
TAutoClass class 291
TAutoCurrencyRef data type specifier 392
TAutoDateRef data type specifier 392
TAutoDouble data type specifier 391
TAutoDoubleRef data type specifier 392
TAutoFloat data type specifier 391
TAutoFloatRef data type specifier 392
TAutoInt data type specifier 391
TAutoIterator class 291
TAutoLong data type specifier 391
TAutoLongRef data type specifier 392
TAutoObject template 387, 391
TAutoObjectByVal template 392, 401
TAutoObjectDelete template 392
TAutoProxy class 408
  *See also* proxy classes
  base for proxy classes 409
  instances, creating 412
  members
    Bind 412
    Unbind 412
TAutoShort data type specifier 391, 393
TAutoShortRef data type specifier 392
TAutoString data type specifier 387, 391
TAutoVal class 391
TAutoVoid data type specifier 392
TBitmap class 221
  accessing 222
  constructing 221
  convert to BITMAP 222
  extending 224
  members
    BitsPixel 223
    Create 224

GetBitmapBits 223
GetBitmapDimension 223
GetObject 222, 223
HBITMAP operator 222
Height 223
Planes 223
SetBitmapBits 223
SetBitmapDimension 223
ToClipboard 223
Width 223
TBitmapGadget class 179
constructing 179
destroying 179
members
MapUIColors 180
SelectImage 179
SysColorChange 180
selecting a new image 179
TBool 391
TBorders structure 173
TBorderStyle enum 173
TBrush class 215
accessing 216
constructing 215
convert to LOGBRUSH structure 216
members
GetObject 216
HBRUSH operator 216
UnrealizeObject 216
reset origin of brush object 216
TButton class 152
TButtonGadget class 80, 180
accessing button gadget information 181
command enabling 182
constructing 180
corner notching 181
destroying 181
members
CommandEnable 182
GetButtonState 181
GetButtonType 181
SetAntialiasEdges 181
SetButtonState 181
SetNotchCorners 181
SetShadowStyle 181
SysColorChange 182
setting button gadget style 181
TButtonGadgetEnabler class 51
TCalc example class 386
TCheckBox class 152
TChooseFontDialog class TData structure 110
TClientDC class 206
TComboBox class 162
TCommandEnabler class 50, 51
constructors 52
enums 52, 57
members
Enable 52, 54
GetHandled 53
Handled 53
HWndReceiver 53
Id 53
IsReceiver 53
SetCheck 52, 57
SetText 52, 55
TControl class 144, 246
TControlBar class 80, 189
constructing 189
TControlGadget class 182
constructing 182
destroying 182
TCreatedDC class 206
TCursor class 231
accessing 232
constructing 231
convert to ICONINFO 232
members
GetIconInfo 232
HCURSOR operator 232
TData nested class 107, 112
constructing dialog boxes 109, 110, 111, 115
members
Error 108
Flags 108, 113
TDC class 206
constructors 207
destructor 207
members
AngleArc 212
Arc 212
BeginPath 211
BitBlt 212
CheckValid 213
Chord 212
CloseFigure 211
DPtoLP 210
DrawFocusRect 212
DrawIcon 212
DrawText 212
Ellipse 212
EndPath 211
EnumFontFamilies 211
EnumFonts 211
EnumMetaFile 210
ExcludeClipRect 210
ExcludeUpdateRgn 210
ExtFloodFill 212
ExtTextOut 212
FillPath 211
FillRect 212
FillRgn 212
FlattenPath 211
FloodFill 212
FrameRect 212
FrameRgn 212
GetAspectRatioFilter 211
GetAttributeHDC 213
GetBkColor 210
GetBkMode 210
GetBoundsRect 210
GetCharABCWidths 211
GetCharWidth 211
GetClipBox 210
GetClipRgn 210
GetCurrentPosition 211, 212
GetDCOrg 208
GetDeviceCaps 208
GetDIBits 212
GetFontData 211
GetHDC 213
GetMapMode 210
GetNearestColor 209
GetPixel 212
GetPolyFillMode 210
GetROP2 210
GetStretchBltMode 210
GetSystemPaletteEntries 209
GetSystemPaletteUse 209
GetTextColor 210
GetViewportExt 210
GetViewportOrg 210
GetWindowExt 210
GetWindowOrg 210
GrayString 212
Handle 213
HDC operator 207
Init 213
IntersectClipRect 210
InvertRect 212
InvertRgn 212
LineDDA 212
LineTo 212
LPtoDP 210
MaskBlt 213
ModifyWorldTransform 210
MoveTo 210, 212
OffsetClipRgn 210
OffsetViewportOrg 210
OffsetWindowOrg 210
OrgBrush 213
OrgFont 213
OrgPalette 213
OrgPen 213
OrgTextBrush 213
OWLFastWindowFrame 213
PaintRgn 212

PatBlt 212
PathToRegion 211
Pie 212
PlayMetaFile 210
PlayMetaFileRecord 210
PlgBlt 213
PolyBezier 212
PolyBezierTo 212
PolyDraw 212
Polygon 212
Polyline 212
PolylineTo 212
PolyPolygon 212
PolyPolyline 212
PtVisible 210
RealizePalette 209
Rectangle 212
RectVisible 210
ResetDC 208
RestoreBrush 209
RestoreDC 208
RestoreFont 209
RestoreObjects 209
RestorePalette 209
RestorePen 209
RestoreTextBrush 209
RoundRect 212
SaveDC 208
ScaleViewportExt 210
ScaleWindowExt 210
ScrollDC 212
SelectClipPath 211
SelectClipRgn 210
SelectObject 208
SelectStockObject 209
SetBkColor 210
SetBkMode 210
SetBoundsRect 210
SetDIBits 212
SetDIBitsToDevice 212
SetMapMode 210
SetMapperFlags 211
SetMiterLimit 210
SetPixel 212
SetPolyFillMode 210
SetROP2 210
SetStretchBltMode 210
SetSystemPaletteUse 209
SetTextColor 210
SetViewportExt 210
SetViewportOrg 210
SetWindowExt 210
SetWindowOrg 210
SetWorldTransform 210
ShouldDelete 213
StretchBlt 212
StretchDIBits 212
StrokeAndFillPath 211
StrokePath 211
TabbedTextOut 212
TextOut 212
TextRect 212
UpdateColors 209
WidenPath 211
TDecoratedFrame class 78, 305
constructing 79
decorating 80
members
Insert 80
TDecoratedMDIFrame class 305
TDesktopDC class 206
TDialog class 97
constructors 98
members
CloseWindow 100, 102
CmCancel 100, 102
CmOk 100, 102
Create 99, 101
Destroy 100, 102
ExecDialog 452
Execute 98, 101, 109, 452
HWindow 106
SendDlgItemMessage 104
SetupWindow 106
UpdateData 114
overriding members 100, 102
TDib class 232
accessing internal structures 234
constructing 233
destroying 233
DIB information 235
members
BITMAPINFO operator 234
BITMAPINFOHEADER operator 234
ChangeModeToPal 235
ChangeModeToRGB 235
FindColor 236
FindIndex 236
GetBits 234
GetColor 236
GetColors 234
GetIndex 236
GetIndices 234
GetInfo 234
GetInfoHeader 234
HANDLE operator 234
Height 235
IsOK 235
IsPM 235
MapColor 236
MapIndex 237
MapUIColors 237
NumColors 235
NumScans 235
SetColor 236
SetIndex 236
Size 235
StartScan 235
ToClipboard 235
TRgbQuad * operator 234
Usage 235
Width 235
WriteFile 235
type conversions 234
TDibDC class 206
TDocManager class 125
constructors 126
handling events 127
members
CreateAnyDoc 126
CreateAnyView 126
FlushDoc 132
GetApplication 443
TDocTemplate class
members
ClearFlag 125
GetDefaultExt 125
GetDescription 125
GetDirectory 125
GetFileFilter 125
GetFlags 125
IsFlagSet 125
SetDefaultExt 125
SetDescription 125
SetDirectory 125
SetFileFilter 125
SetFlag 125
TDocument class 305
constructors 129
destructor 131
members
AttachStream 130
CanClose 132
Close 130
Commit 131
DetachStream 130
FindProperty 141
GetDocManager 126, 132
InStream 130
IsDirty 131
IsOpen 131
NotifyViews 132, 133
Open 130
OutStream 130
PropertyFlags 141
PropertyName 141
QueryViews 132, 133
Revert 131
SetDocManager 132
overriding members 129
protecting data 131
TEdge enum 71

TEdgeConstraint class 70
members
Above 72
Absolute 73
Below 72
LeftOf 72
PercentOf 73
RightOf 72
SameAs 73
Set 70, 72
TEdgeOrHeightConstraint class 70
TEdgeOrWidthConstraint class 70
TEdit class
members
SetValidator 242
TEditFile class 113, 446
adding client windows 447
TEditSearch class 113, 446
adding client windows 447
TEditWindow class 446
template classes 120
constructing 123
instantiation 122, 123-124
templates
document 117, 121-125
changing 125
creating 120
designing classes 121-122
registration tables 122-123
properties, setting 125
temporary dialog boxes 97
text, replacing 113
text-based applications
find-and-replace operations 113
TextOut member function
TDC 212
TextRect member function
TDC 212
TFileDialog class 448
TFileDocument class 305
TFileOpenDialog class 448
TFileWindow class 446
TFindDialog class 114, 448
TFindReplaceDialog
TData structure 113
TFloatingFrame class 192
TFont class 217
accessing 218
constructing 217
convert to LOGFONT structure 218
members
GetObject 218
HFONT operator 218
TFontListBox class 144
TFrameWindow class 75, 305, 431
changing frame windows 78
constructing 66, 76, 77
constructing dialog boxes 98
members
AddWindow 77
AssignMenu 78, 89, 435
Attr 89, 435
GetMenuDescr 78
Init 66
IsFlagSet 77
MergeMenu 78
RestoreMenu 78
SetIcon 78
SetMenu 78
SetMenuDescr 78, 94
TGadget class 171
cleaning up 175
constructing 171
derived classes 177
deriving from 175
destroying 172
initializing 175
members
Clip 174
CommandEnable 175
GadgetSetCapture 177
GetBorders 173
GetBorderStyle 173
GetBounds 173
GetDesiredSize 174
GetEnabled 175
GetId 172
GetInnerRect 176
GetMargins 173
Inserted 175
Invalidate 176
InvalidateRect 176
LButtonDown 177
LButtonUp 177
MouseEnter 177
MouseLeave 177
MouseMove 177
Paint 176
PaintBorder 175
PtIn 176
Removed 175
SetBorders 173
SetBorderStyle 173
SetBounds 173
SetEnabled 175
SetMargins 173
SetShrinkWrap 173
SetSize 174
SysColorChange 174
TrackMouse 177
Update 176
WideAsPossible 174
mouse events 176
painting 175
TBorders structure 173
TMargins structure 173
TGadgetWindow class 182
capturing mouse movements for gadgets 186
constructing 183
converting 188
creating 183
derived classes 189
deriving from 187
destroying 183
determining size 183
idle action processing 187
layout units 188
members
CommandEnable 187
Create 183
FirstGadget 187
GadgetChangedSize 185
GadgetFromPoint 187
GadgetReleaseCapture 186
GadgetSetCapture 186
GadgetWithId 187
GetDesiredSize 188
GetDirection 184
GetFont 186
GetFontHeight 186
GetHintMode 186
GetInnerRect 188
GetMargins 188
IdleAction 187
Insert 183
LayoutSession 184
LayoutUnitsToPixels 188
NextGadget 187
Paint 187
PaintGadgets 188
PositionGadget 185
Remove 184
SetDirection 184
SetHintCommand 186
SetHintMode 186
SetMargins 184
SetShrinkWrap 185
TileGadgets 185
message response functions 189
painting 187
shrink wrapping 185
TGauge class 159
TGdiObject class 203
TGroup enum 92
THintMode enum 186

this pointer 98
dialog boxes 100
This variable 403
throw keyword 61
THROW macro 61, 64
Throw member function
TXBase 61
throwing exceptions 61
THROWX macro 64
THSlider class 158
TIC class 206
Tic Tac Toe example 295, 375
registering DLL server 379
TIcon class 229
accessing 230
constructing 229
convert to ICONINFO 231
members
GetIconInfo 231
HICON operator 230
TileGadgets member function
TGadgetWindow 185
timers, DLL servers 374
TInputDialog class 107
TInStream class 130
TLayoutMetrics class 70
TLayoutWindow class 69
constructing 74
defining constraints 70, 73, 74
members
EvSize 75
GetChildLayoutMetrics 75
Layout 74
RemoveChildLayout-
Metrics 75
SetChildLayoutMetrics 75
TLB file extension 301
TListBox class 144, 145
members
AddString 147
TListBoxData class 166
TLocaleString class 399
TLocation enum 80
TLookupValidator class 240
TMargins structure 173
TMDIChild class 82, 439
TMDIClient class 439
creating MDI child
windows 83-84
manipulating MDI child
windows 83
members
CmCreateChild 83
InitChild 83
TMDIFrame class 82, 305, 439
members
ActiveChild 441
GetActiveChild 441
TMeasurementUnits enum 72
TMemoryDC class 206
TMenu class 85
constructors 85
TMenuDescr class 85, 92, 325
constructors 93
grouping pop-up menus 93
merging menus 94
TMenuItemEnabler class 51
TMessageBar class 190
constructing 190
destroying 190
members
SetHintText 190
SetText 190
setting
hint text 190
message bar text 190
TMetaFileDC class 206
TModeIndicator enum 191
TModule class 15, 258, 259
getting the module
instance 452
TOcApp class
binding to window 290
connector object 288
created in factories 364
members
SetOuter 365
SetupWindow 370
TOcAppMode enum 364, 379
TOcAutoFactory template 385
TOcDocument class 300, 333
members
Close 334, 368
ToClipboard member function
TBitmap 223
TDib 235
TPalette 220
TOcModule class 317
base for application
object 304
members
IsOptionSet 379
TOcPart class 299
*See also* parts
connector object 288
getting coordinates from 335
members
Draw 336
GetRect 335
TOcRegistrar class 300
*See also* registrar objects
members
constructor 307, 310, 331,
363
Run 362, 363
TRegistrar vs. 385
TOcRemView class 300, 302
*See also* remote views
created by
TOleWindow::Create-
OcView 357
members
ReleaseObject 368
SetOuter 365
SetupWindow 367
servers 367
non-embedded 368
TOcView class 302, 321, 333
*See also* views
members
ReleaseObject 334
SetupWindow 334
ToggleModeIndicator member
function, TStatusBar 192
toggles, command-enabling
objects 52, 56
TOleAllocator class
*See also* memory allocator
automation controllers 408
TOleDocument class 305
compound documents
and 312
members
Commit 312, 313
CommitTransactedStorage
313
Open 312, 313
PreOpen 313
transacted mode, default 313
TOleDocViewFactory
template 306, 310, 348
TOleFactory template 318, 356
TOleFrame 305
TOleFrame class 305, 320
members
AddUserFormatName 309
EvAppBorderSpaceSet
312, 328
OLE event handlers 322
SetupWindow 290
TOleMDIFrame class 305, 320
members
OLE event handlers 322
TOleView class 305, 312
members
InvalidatePart 350

TOleWindow class 305
members
constructor 321
CreateOcView 321
OLE event handlers, table of 322
SelectEmbedded 324
tool bars
merging 312
ObjectWindows containers, in 328
standard identifier for 312
tool boxes 192
Top enum 80
ToPage data member
TData 115
TOpenSaveDialog class
TData structure 111, 112
Touches member function
TRegion 226
TOutStream class 130
TPaintDC class 206
TPalette class 218
accessing 219
constructing 218
extending 220
extract number of table entries 219
members
AnimatePalette 220, 221
Create 220
GetNearestPaletteIndex 219
GetNumEntries 220
GetObject 219
GetPaletteEntries 220
GetPaletteEntry 220
HPALETTE operator 219
ResizePalette 220
SetPaletteEntries 220
SetPaletteEntry 220
ToClipboard 220
UnrealizeObject 220
TPen class 213
accessing 215
constructing 213
convert to LOGPEN structure 215
members
GetObject 215
HPEN operator 215
TPlacement enum 184
TPopupMenu class 85, 88
members
TrackPopupMenu 88
TPrintDC class 206
TPrintDialog class
TData structure 115
TPrinter class 195
constructor 196
members
Print 199
SetPrinter 201
Setup 201
TPrinterDialog class 201
TPrintout class 195, 197
members
BeginDocument 200
BeginPrinting 200
EndDocument 200
EndPrinting 200
GetDialogInfo 199
HasNextPage 197
HasPage 200
PrintPage 195, 197, 200
SetPrintParams 199
TPXPictureValidator class 241
TrackMouse data member
TGadget 177
TrackPopupMenu member function
TPopupMenu 88
TRangeValidator class 240
transacted mode 313
transfer buffers 164
TransferData class 169
translating
logical coordinates to physical coordinates 210
physical coordinates to logical coordinates 210
TRect structure 133
TRegion class 224
accessing 226
constructing 224
destroying 224
members
!= operator 227
&= operator 228
+= operator 227
-= operator 228
= operator 227
== operator 227
^= operator 229
|= operator 228
Contains 226
GetRgnBox 226
HRGN operator 226
SetRectRgn 226
Touches 226
TRegistrar class 300
*See also* registrar objects
members
constructor 385
Run 311, 385
TOcRegistar vs. 385
TRegItem class
in registration structures 372
TRegLink class
*See also* document lists
C++ servers, in 362
CreateOleObject, in 358
explained 358
TOleWindow::CreateOcView, and 357
TRegList class 122
built in registration table 300
in registration structures 372
TReplaceDialog class 114, 448
TRgbQuad * (TDib) operator 234
troubleshooting 452
TRY macro 64
TScreenDC class 206
TSearchDialog class 448
TSeparatorGadget class 178
TServedObject 291
TServedObject class
automation connector object 291
TShadowStyle enum 181
TSlider class 158
TSpacing structure 192
TStatic class 150
TStatusBar class 80, 191
constructing 191
displaying mode indicators 191
inserting gadgets 191
members
Insert 191
SetModeIndicator 192
SetSpacing 192
ToggleModeIndicator 192
spacing gadgets 192
TStorageDocument class 312
TStringLookupValidator class 241
TSystemMenu class 85, 87
TTextGadget class 178
accessing text 178
constructing 178
destroying 178
members
GetText 178
SetText 179
TTileDirection enum 183, 193
TToolBar class
changing tool box dimensions 193
TToolBox class 192
constructing 192
members
SetDirection 193

TTypeLibrary interface 291
TUIHandle class 271
TUnknown class 288
Turbo Debugger for Windows
debugging DLL servers 377
turning off
command-enabling
objects 52, 53-55
exceptions 64
TValidator class 240
members
Error 244
IsValid 243
IsValidInput 243
Valid 243
TVbxControl class 246
constructors 247
members
AddItem 254
GetEventIndex 251
GetEventName 251
GetNumEvents 251
GetNumProps 252
GetProp 252
GetPropIndex 252
GetPropName 252
Move 254
Refresh 254
RemoveItem 254
SetProp 253
TVbxEventHandler class 246, 249
members
EvVbxDispatch 249
mixing with interface
classes 246
TView class 305
adding menus to views 135
closing views 136
constructors 134
displaying data 135-136
handling events 135, 136
members
FindProperty 141
GetViewName 135
GetWindow 135
PropertyFlags 141
SetDocTitle 135
overriding members 134, 135
TVSlider class 158
TWidthHeight enum 71
TWindow class 144, 305, 431
as generic interface object 30
child-window attributes 67
constructing 66
converting from
TWindowsObject 431
creating interface elements 68
creating the main window 68
members
Attr 67
CanClose 102
CloseWindow 445
Create 68, 101, 183, 451
Destroy 445
DisableAutoCreate 100
DrawMenuBar 86
EnableAutoCreate 100
GetApplication 17, 443
Init 66
Invalidate 109
MakeWindow 451
Paint 195, 199, 444
SetupWindow 145, 146
ShowWindow 99, 101
ShutDownWindow 445
TWindowAttr structure 67
TWindowDC class 206
TWindowsObject class
converting to TWindow 431
members
AfterDispatchHandler 448
BeforeDispatchHandler
448
DispatchAMessage 449
FirstThat 445
ForEach 445
GetModule 443
TXAuto class 60
TXBase class 59, 60-61
constructing 60
deriving from 61
destructor 60
members
Clone 60
InstanceCount 60
Throw 61
TXCompatibility class 60
TXGdi class 63
TXInvalidMainWindow class 63
TXInvalidModule class 63
TXMenu class 63
TXOle class 60
TXOutOfMemory class 60, 62
TXOwl class 59, 61
constructing 62
deriving from 62
destructor 62
members
Clone 62
TXPrinter class 63
TXValidator class 63
TXWindow class 63
TXWindow exception 102
type libraries 405-406
AutoGen, used by 408
browsing example 295
controller, used by 408
defined 301
file extensions 301
Help files, and 405
type substitution 205
typehelp key 384, 390, 400, 406
-TypeLib option 301, 349, 405
typographic conventions 3

## U

Unbind member function
TAutoProxy 412
Unchecked enum 52, 57
undoing automation
commands 387
unexpected user responses 59
universally unique identifier *See* GUID
UnrealizeObject member
function
TBrush 216
TPalette 220
unregistering
OLE applications 349
with Register utility 379
-UnregServer option 349
untitled applications 18
Update member function
TGadget 176
update rectangle functions 210
update region functions 210
UpdateColors member function
TDC 209
UpdateData member function
TDialog 114
updating a gadget 176
updating menu bars 86
usage key 383, 384
Usage member function
TDib 235
user input 107
processing 23, 25
dialog boxes and 99, 102
unexpected 59
user interfaces
user interfaces (OLE)
*See also* interface objects
activating 271
Convert command 272
described 267-275
DLL servers and 375
inactive objects 271
in-place editing 269
linking 273

ObjectWindows
container 322
open editing 274
overriding event
handlers 322
selecting 271
verbs 272
utility programs
ObjectComponents 292
UUID (universally unique
identifer) *See* GUID


## V


Val variable 388
Valid member function
TValidator 243
validating automation
arguments 387
validators 239, 242
abstract 240
constructing 242
error handling 244
filter 240
linking to edit controls 242
lookup 240
overriding member
functions 243
picture 241
range 240
standard 239
string lookup 241
TFilterValidator 240
TLookupValidator class 240
TPXPictureValidator
class 241
TRangeValidator class 240
TStringLookupValidator
class 241
TValidator class 240
variables
DLLs and multiple
processes 342, 353
MainWindow 441
VARIANT unions 291, 387
VBX controls 245
accessing 251
classes 246
control methods 253
event handling 246
event response table 249
finding event
information 251
finding property
information 252
getting control properties 252
handling messages 249
implicit and explicit
construction 248
interpreting a control
event 250
properties 251
VBXEVENT structure 249, 250
VBXInit function 245
VBXTerm function 245
verb*n* key 400
verbs (OLE)
defined 301
Quattro Pro 272
user interface for 272
version key 308, 384
Vertical enum 183, 193
view classes 117, 120, 134-136
accessing data 134
implementing 134
interface objects and 135-136
view events, handling 335
view objects 119-120
adding menus 135
closing 136
constructing 134
displaying data 135-136
handling events 135, 136,
138-139
releasing 334
response tables 132, 138
setting properties 139-142
view windows
*See also* client windows
binding to TOcView
object 334
C++ containers 332
C++ servers 366
CPPOCF1 example 332
viewport mapping functions 210
views 132
defined 301
ViewWnd_PaintParts
procedure 336
virtual bases
constructing 435
downcasting to derived
types 435
virtual functions 8
document classes 129
overriding 129
view classes 134-135
VN_DEFINE macro 138


## W


warnings, compiler 453
WB_MDICHILD flag 440
WEP function 256
wfAlias flag 77
wfAutoCreate flag 100
WideAsPossible data member
TGadget 174
WidenPath member function
TDC 211
widgets 11
Width member function
TBitmap 223
TDib 235
WIN30, defining 452
WIN31, defining 452
window classes 10
base class 10
window constructors
converting 431
window handles 31
window mapping functions 210
window margins 184
window objects 65
constructing 65-66
constructing dialog boxes 98
creating main windows 68
creation attributes 66-68
overriding 67
decorated frame
windows 78-80
constructing 79
decorating 80
defining constraints 74
frame windows 75-78
constructing 76-78
menu objects 89-95
interface elements and 65, 68
layout windows 69-75
defining constraints 70, 73
MDI windows 80-84
style attributes 32
WindowGroup enum 92
windows 10
*See also* window objects
child 34, 67, 70, 101
MDI applications 81, 83
creating 83-84
client 98
MDI applications 83
contents, printing 198
decorated 10
destroying 36
main *See* main windows
maximizing 24
MDI 10
parent 100
dialog boxes and 98, 100
Windows applications 27
32-bit executables 21
initializing 21

closing 15, 26-27
colors, setting 109
controls 11
creating 16
encapsulating 15
instantiation 19, 21, 23
messages 25
naming 16, 18
opening files 111
printing data 115
replacing text 113
running 16, 17
multiple copies 21, 23
saving files 112
searching for text 113
setting fonts 110
Windows messages 41, 45
ObjectComponents applications, in 322
Windows NT applications 25
WinMain
automation servers 385
C++ containers 330
C++ servers 362
WinMain function
calling 19
constructing applications 20
parameters 15, 24
passing 20
WinRun utility 293
WM_ACTIVATEAPP 322
WM_COMMAND 323
WM_COMMAND_ENABLE event 49
WM_COMMAND_ENABLE message 49-50, 182
WM_COMMANDENABLE 323
WM_DROPFILES 323
WM_INITMENUPOPUP message 50
WM_LBUTTONDBLCLK 322
WM_LBUTTONDOWN 322
WM_LBUTTONUP 322
WM_MDIACTIVATE 322
WM_MOUSEACTIVATE 322
WM_MOUSEMOVE 322
WM_OCEVENT 289, 334
C++ servers, in 370
containers
C++ 334
ObjectWindows 322
sent by application and view connectors 281
servers, C++ 368
WM_PAINT 323
WM_RBUTTONDOWN 322
WM_SETCURSOR 322
WM_SETFOCUS 322
WM_SIZE 322
WM_SIZE message 75, 322
WM_SYSCOLORCHANGE message 182
WM_TIMER 322
WM_VBXFIREEVENT macro 249
word-processing programs
view objects vs. 119
working with device contexts 206
WPARAM parameter 429
WriteFile member function
TDib 235
WS_VISIBLE flag 99, 101
WS_VISIBLE style 31

## X

XEND resource delimiter 397
XLAT resources 299, 350, 397
AutoCalc example 400
explained 399
xmsg class, initializing 60

**Borland**

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Brazil, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1245WW21776 • BOR 7774