ObjectWindows® for C++

User's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

Copyright © 1991 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Windows, as used in this manual, shall refer to Microsoft's implementation of a windows system. Other brand and product names are trademarks or registered trademarks of their respective holders.

R1

10 9 8 7 6 5 4 3 2

C O N T E N T S

T A B L E S

F I G U R E S

INTRODUCTION

ObjectWindows provides an exciting new way to develop applications for Microsoft Windows. Until recently, most Windows programming required the Microsoft C compiler and a lot of separate, complex development tools. As a result, developing Windows programs tended to be long, complicated, and confusing. With the ObjectWindows application framework, Windows programming has become much easier.

Programming in Windows introduces many new wrinkles you may never have had to think about before, such as dealing with text and graphics in resizeable windows, interacting with other programs in a multitasking environment, and manipulating the nearly 600 functions in the Windows Application Programming Interface (API). Perhaps the most frustrating part of all can be figuring out just which basic things your program needs to do to function as a Windows application and then making sure you've done all of them.

ObjectWindows lets you get around all that. It is an object-oriented class library that encapsulates the behaviors (application-level and window-level) that Windows applications commonly perform. ObjectWindows eases Windows development by providing

- a consistent, intuitive, and simplified interface to Windows
- supplied behavior for window management and message-processing
- a basic framework for structuring a Windows application

You *inherit* this base functionality, which leaves you free to concentrate your efforts on the unique requirements of your application.

ObjectWindows does not provide classes to entirely encapsulate all the logical entities of a Windows application. But, by reusing the classes that ObjectWindows does provide, you'll significantly

reduce application development time, and have far less code to maintain.

What's in the ObjectWindows manual

Because ObjectWindows uses some new techniques, this book includes a lot of explanatory material. It has three parts:

- Part 1, *Learning ObjectWindows*, introduces the principles of writing Windows applications. It includes a tutorial that walks you through the process of writing and extending an ObjectWindows application.
- Part 2, *Using ObjectWindows*, gives greater detail on the elements of ObjectWindows itself. It includes an overview of the class hierarchy and explains how it interacts with the Windows environment.
- Part 3, *ObjectWindows Reference*, gives reference material on ObjectWindows classes, ObjectWindows types, ObjectWindows constants, Windows functions, Windows types, and Windows constants. This material is arranged alphabetically with thumbtabs for easy access.

Hardware and software requirements

Because ObjectWindows is an application framework for writing Windows applications, the basic hardware requirements for ObjectWindows applications are the same as those for Windows:

- a hard disk
- 2MB of memory or more
- Windows-compatible graphics display
- Windows 3.0 or later in standard or 386 enhanced mode

You can compile ObjectWindows applications (and the ObjectWindows source code itself) with Borland C++ or Turbo C++ for Windows.

PART 1

Learning ObjectWindows

C H A P T E R

1

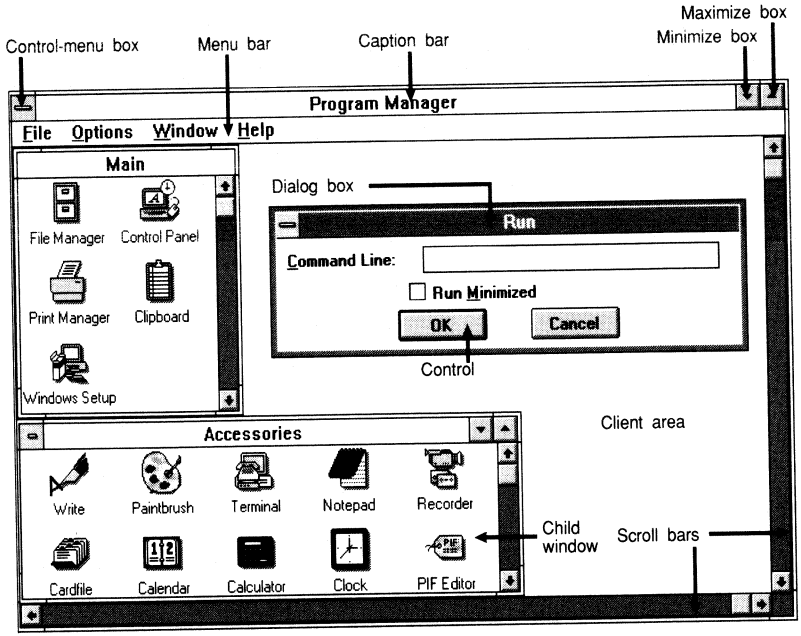
Inherit the window

This chapter is an overview of programming for Microsoft Windows, with an emphasis on object-oriented programming. You will learn what goes into a Windows application and what comes out. You will learn the required behavior of a Windows application, and how object-oriented programming with ObjectWindows automates many tasks and simplifies others.

What is a Windows application?

Figure 1.1 shows the major components of a Windows application. To successfully understand and use ObjectWindows, you should be familiar with these components.

Figure 1.1
The onscreen components of a Windows application



A Windows application is a special type of program that

- must be in a special executable .EXE file format.
- runs only under Windows.
- generally runs in a rectangular window on the screen.
- follows user interface guidelines to appear and perform in a standard way.
- can run simultaneously with other Windows and non-Windows applications, including other instances of itself.
- can communicate and share data with other Windows applications.

Benefits of Windows

Windows offers many benefits to both users and developers. Benefits to users include

- standard and predictable operation. If you know how to use one Windows application, you know how to use them all.
- no need to set up devices and drivers for each application. Windows provides drivers to support peripherals.
- interprocess cooperation and communications.

- multitasking: the ability to run many applications at once.
- access to more memory. Windows supports protected mode on the 80286, 80386, and i486; it supports virtual memory on the 80386 and i486.

Benefits to developers include

- device-independent graphics, so graphical applications run on all standard display adapters.
- immediate support for a wide range of printers, monitors, and pointing devices such as mice and trackballs.
- a rich library of graphics routines.
- more memory for large programs.
- support for menus, icons, bitmaps, and more.

Requirements

The array of benefits offered by Windows requires a fairly sophisticated list of hardware: Windows generally requires better graphics hardware, more memory, and faster processors for equivalent performance compared to a DOS application. If you have an 80286 machine or better and at least 2MB of memory, Windows will run fine.

Object-oriented windowing

Programming for a windowing environment requires familiarity with many new concepts. Developing even a simple Windows program can be a daunting task. ObjectWindows simplifies the process, allowing you to focus on your application's function, rather than its form.

ObjectWindows' framework lets you use objects to represent the fairly complex elements of a Windows program. ObjectWindows window objects encapsulate data that all windows require, perform common window operations, and respond to common Windows messages and events. ObjectWindows window and application classes completely manage the processing of messages, which normally comprises the bulk of a Windows application.

A better interface to Windows

ObjectWindows uses the object-oriented features of Turbo C++ to encapsulate parts of the Windows API, insulating you from the details of Windows programming. As a result, you can write Windows programs with much less time and effort. Specifically, ObjectWindows provides three helpful features:

- encapsulation of window information
- abstraction of many Windows API functions
- automatic message response

Encapsulating Window information

ObjectWindows supplies objects that define behavior and data storage for the windows, dialog boxes, and controls of Windows applications. In an ObjectWindows application, an *interface object* serves as a representative of a visual Windows *interface element*. Although the interface object and interface element work in close partnership, it is important to understand the distinction between the two.

An interface object is associated with a new interface element by calling (indirectly) its **Create** member function. **Create** calls the appropriate Windows function to create a new interface element. Data members of the interface object, passed on the Windows call, define the physical attributes of the interface element to be created. Windows returns the handle, or identifier, of the created interface element. The interface object stores this handle as its *HWindow* data member.

Windows functions that operate on a window require that the window's handle be passed; storing the handle as a data member makes it easily accessible. Similarly, data members can be used to store drawing tools or status information for a particular window.

Abstracting Windows functions

Windows applications get their appearance and behavior by calling the set of almost 600 functions that make up the Windows API. Each function can take a variety of parameters of many different types. Although you can call any Windows function directly from Turbo C++, ObjectWindows simplifies the task by offering object member functions that abstract many of the function calls.

As noted earlier, many of the parameters for Windows functions are already stored in the data members of interface objects. Thus, member functions can use this data to supply Windows functions with parameters. In addition, ObjectWindows groups related function calls into single member functions that perform higher-level tasks. The result is a streamlined, easier-to-use interface to Windows.

While this approach greatly reduces your dependence on the hundreds of Windows API functions, it does not restrict you from calling the API directly. ObjectWindows offers the best of both worlds: high-level, object-oriented development plus maximum control over the graphic environment.

Automating message response

In addition to telling the Windows environment to do things, most applications need to be able to respond to the hundreds of Windows messages that result from user actions (such as clicking the mouse), other applications, or other sources. Remember that Windows sends your application a message every time the user clicks a mouse button or moves the mouse. Your application might receive thousands of messages in just a few minutes. Processing and responding to messages correctly is critical to the proper functioning of your program.

Objects, with their ability to inherit and redefine behavior (member functions), are perfectly suited to the task of responding to incoming stimuli (Windows messages). ObjectWindows takes Windows messages and turns them into Turbo C++ member function calls. Therefore, using ObjectWindows, you simply define a member function to respond to each message your application needs to handle. For example, when the user presses the left mouse button, Windows generates a WM_LBUTTONDOWN message. If you want a window or control in your program to respond to such mouse clicks, you simply define a **WMLButtonDown** member function keyed to the WM_LBUTTONDOWN message. Then, whenever Windows sends that message, your object will automatically call the member function you've defined.

Such member functions are called *message response functions*. You key a particular Windows message to a message response function using a feature of Borland's C++ compilers called *dynamic dispatch virtual tables* (DDVTs). When you declare a *dynamically dispatchable member function*, you also specify an integer *dispatch index*. For example, in

```
virtual void Test() = [100];
```

Test is defined as a class's dynamically dispatchable member function with a dispatch index of 100.

If you examine windows.h, you'll see that Windows messages are given descriptive names like WM_LBUTTONUP, but are actually just integer values, so you can use DDVTs with Windows messages.

An ObjectWindows application directs a Windows message to the function whose dispatch index equals the value of the message. For example,

```
virtual void WMLButtonUp(RTMessage Msg) = [WM_FIRST + WM_LBUTTONUP];
```

The sum of the constants WM_FIRST and WM_LBUTTONUP is the integer that Windows uses to identify a message.

declares a void function that responds to the Windows message WM_FIRST + WM_LBUTTONUP. There's nothing else to do—ObjectWindows knows when you've added a new message response function and automatically starts dispatching messages to it. ObjectWindows expects message response functions to take an **RTMessage** parameter type. Remember that an **RTMessage** is a reference to a **TMessage**. **TMessage** objects contain all the information that Windows passes along with a message.

There are a few limitations on using dynamically dispatchable member functions:

- When a dynamically dispatchable function in a base class is redefined in a derived class, the same dispatch index must be used. For example, the following is illegal:

```
class TestBase
{
  virtual void sample() = [100];
};

class TestDerived : public TestBase
{
  virtual void sample() = [299];
};
```

This definition is illegal because the dispatch indexes are different.

- Each dynamically dispatchable function in a class hierarchy must have a unique dispatch index. For example, the following is illegal:

This statement is illegal because it attempts to assign two different functions to the same dispatch index.

```
class TestBase
{
  virtual void sample1() = [100];
  virtual void sample2() = [200];

  virtual void sample3() = [100];
};
```

- An "ordinary" (non-dynamically dispatchable) virtual function in a base class cannot be redefined in a derived class by a dynamically dispatchable function. For example, the following is illegal:

This definition is illegal because it's attempting to redefine an ordinary function with a dynamically dispatchable one.

```
class TestBase
{
  virtual void sample();
};

class TestDerived : public TestBase
{
  virtual void sample() = [100];
};
```

- A dynamically dispatchable function in a base class cannot be redefined in a derived class by an "ordinary" virtual function. For example, the following is illegal:

This definition is illegal because it's attempting to redefine a dynamically dispatchable function with an ordinary one.

```
class TestBase
{
  virtual void sample() = [100];
};

class TestDerived : public TestBase
{
  virtual void sample();
};
```

- When a class is derived from more than a single base class, the derived class is said to be multiply inherited. Only the first base class for a multiply-inherited derived class can have dynamically dispatchable functions. For example, the following class definition is allowed:

```
class A
{
  virtual void AA() = [100];
};
```

```
class B
{
};

class C : public A, public B
{
};
```

Here, the dynamically dispatchable function is in the first base class, so it's OK.

- but the following is not:

```
class A
{
};

class B
{
  virtual void BB() = [100];
};

class C : public A, public B
{
};
```

*This definition is illegal because the **second** base class, class **B**, has a dynamically dispatchable member function.*

- When the class object is pointed to by a pointer and no specific scope override is used, you may not call dynamically dispatchable functions. For example, the following is illegal:

```
class Base
{
  virtual void sample() = [100];
};

int main(void)
{
  Base b;
  Base *BasePtr = &b;

  BasePtr->sample();
}
```

This statement is illegal because you can't directly call a dynamically dispatchable member function through a pointer.

- A virtual base class cannot have dynamically dispatchable member functions. For example, the following is illegal:

```
class Base
{
  virtual void sample() = [100];
};
```

This definition is illegal because a virtual base class can't have dynamically dispatchable member functions.

```
class Derived : virtual public Base
{
};
```

The alternative method of responding to Windows messages, which ObjectWindows avoids (but Microsoft C programming requires), is to use a lengthy **switch** statement with a case for *every* Windows message your program is to respond to. The following example is from a Windows application written in C; as you can see, it has **switch** statements within **switch** statements.

This code sample shows the non-object-oriented way of responding to Windows messages.

```
switch (message)
{
  case WM_DDE_CLOSED:
    CloseMsgWnd();
    break;

  case WM_COMMAND:
    switch (wParam)
    {
      case IDM_QUIT:
        // User has selected QUIT from menu.
        PostMessage(hWnd, WM_CLOSE, 0, 0L);
        break;

      case IDM_HOME:
        // Home key was hit.
        SendMessage(hWnd, WM_HSCROLL, SB_TOP, 0L);
        SendMessage(hWnd, WM_VSCROLL, SB_TOP, 0L);
        break;

      case IDM_ABOUT:
        // User has selected ABOUT from menu.
        lpproc = MakeProcInstance(About, hInst);
        DialogBox(hInst, MAKEINTRESOURCE(ABOUT), hWnd, lpproc);
        FreeProcInstance(lpproc);
        break;

        :
```

Obviously, with up to 200 different Windows messages, this **switch** statement would soon become unmanageable. ObjectWindows's use of DDVTs is easier.

The structure of a Windows program

With so many software elements such as DOS, Windows, your application, and other applications interacting at once, it helps to know how parts of your Windows applications interact with the world around them. This section explores the structure of Windows and typical Windows applications written in Turbo C++ with ObjectWindows.

The structure of Windows

At run time, the functionality of Windows and its API resides in three external library modules, which are called by the currently running applications:

- KERNEL.EXE—Responsible for memory and resource management, scheduling, and interaction with DOS.
- GDI.EXE—Responsible for displaying graphics on the screen and printer.
- USER.EXE—Responsible for window management, user input, and communications.

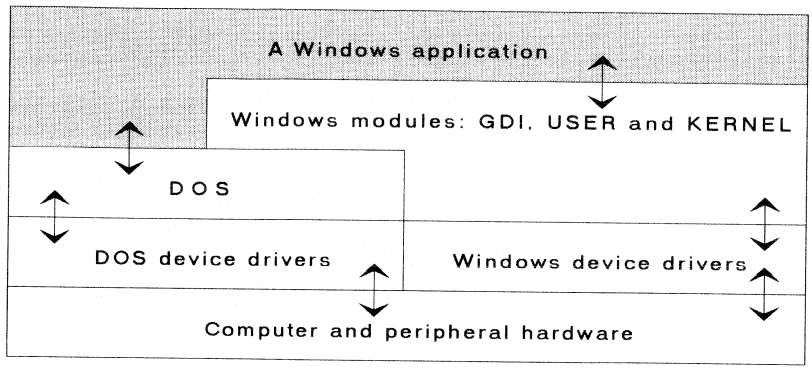
These modules are a part of the retail version of Windows. Assuming your users have Windows, the modules will already be loaded on their computers. You will supply a program that uses these library modules, but does not actually include them.

Interacting with Windows and DOS

Because of the limited scope of the DOS operating system, it's easy to overlook the contribution DOS makes to the successful operation of your DOS application programs. Nonetheless, a DOS program runs because of the interaction between your application code and the facilities of the operating system. The same is true of a Windows program. Because Windows offers so many more operating system functions, it is harder to overlook the interplay between Windows and an application. Your program must continually interact with the operating system (DOS plus Windows).

Figure 1.2
How Windows applications interact with Windows and DOS.

The shaded part is what you write.



"Hello, Windows"

The traditional way to introduce a new language or environment is to present a "Hello, World" program written in the language or for the environment. This program usually consists of only enough code to display the string "Hello, World" on the screen.

Of course, in Windows there's a lot more to do than that. You need to display a window, write in it, then make the window understand how to interact with the world around it, at least enough for you to close the window and make it go away. If you do this from scratch, it takes quite a bit of code just to get those basic tasks done.

That's because Windows has a list of requirements an application must meet before it can run in Windows. Even the simplest program requires a substantial amount of code. Fortunately, programs written with ObjectWindows automatically meet most of those requirements, including creating and displaying the main window. Therefore, *HelloWorld* is simplified to just the following:

This is HELLOAPP.CPP.

```
#include <owl.h>

// Define a class derived from TApplication
class THelloApp :public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
            hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};
```

```
// Construct the THelloApp's MainWindow data member
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  THelloApp HelloApp ("HelloApp", hInstance, hPrevInstance,
                      lpCmdLine, nCmdShow);
  HelloApp.Run();
  return HelloApp.Status;
}
```

Application startup responsibilities

Windows passes four parameters to an application when it begins execution. In an ObjectWindows program, these parameters are passed to the constructor of your application object, derived from the **TApplication** class. These parameters are also stored in data members of the application object so they can be conveniently accessed from your code.

These application data members are described here:

- *hInstance* holds a handle to the application instance.
- *hPrevInstance* holds a handle to the previous instance of the same application. It's zero if this is the first executing instance.
- *lpCmdLine* holds an **LPSTR** to a string representing the application startup command line, including file-name options. For example, "CALC.EXE /M" or "WORDPROC.EXE LETTER1.DOC."
- *nCmdShow* holds an integer representing the initial display mode for the main window. It's used for calls to the **Show** member function.

The first thing an ObjectWindows application must do is construct an application object from the base **TApplication** class. In this example, we use **THelloApp**.

For the first instance of an ObjectWindows application, **Run** calls **InitApplication** to perform needed initializations for the whole application. **Run** initializes all instances (including the first) by calling **InitInstance**. **InitInstance** then calls the **InitMainWindow** member function to construct and initialize the main window object.

Finally, **Run** starts the main message loop of your ObjectWindows application. **Run** returns only when the program terminates. The exit status of the application is stored in the *Status* data member, which you should return to Windows as we do in the **WinMain** function, shown previously.

Main window responsibilities

The main window of an application is the window that first appears when an application is started. It is responsible for presenting to the user a list of available commands (a menu). During the course of the application session, the main window also manages the application's interface, and in many cases, serves as the program's only working area. Other, more complex applications might have many windows that serve as work areas. Finally, when the user closes the main window, it is responsible for initiating the process to close the application.

The application development cycle

Since there are certain requirements of any Windows application (initializing the main window, for one), it is usually easiest to begin your application by using an existing Windows application and customizing it from there. ObjectWindows supplies many sample programs; choose the one most like your application. Then you can begin the programming cycle familiar to many:

1. Create the new program code.
2. Create resources for menus, dialog boxes, and so on.
3. Compile the program.
4. Debug the program.

C H A P T E R

2

Stepping through Windows

Now that you've been introduced to the ObjectWindows library, you're ready to start building some simple ObjectWindows programs. In the next few chapters, you'll learn how to build a graphical, interactive Windows program, complete with menus, file saving and loading, graphics and text drawing, and even a simple help system. Along the way, you'll be introduced to the major principles of Windows application design, such as message processing, managing parent and child window relationships, and automatic graphics redrawing.

This walk-through consists of ten steps, described in Chapters 2 through 6:

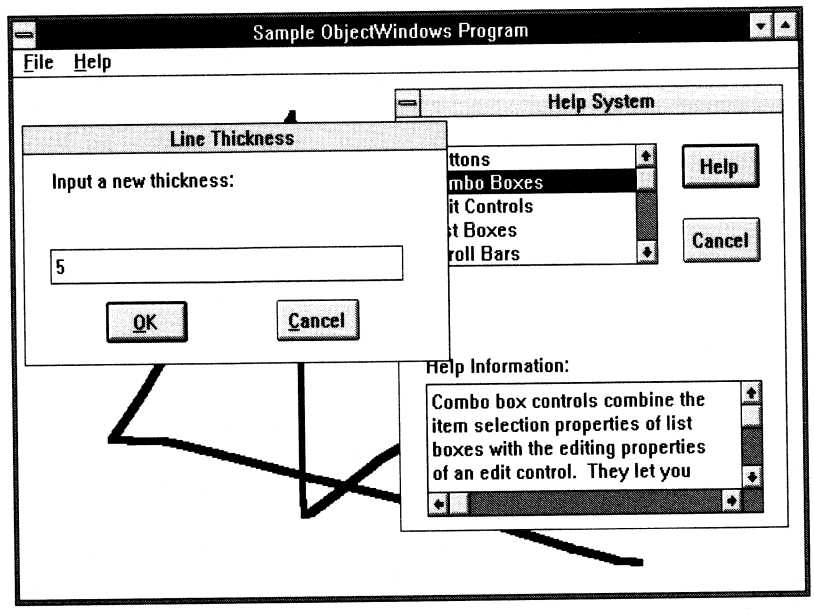
- Step 1: A simple Windows application
- Step 2: The main window class
- Step 3: Drawing text in a window
- Step 4: Drawing lines in a window
- Step 5: Changing the line thickness
- Step 6: Painting graphics
- Step 7: A menu for the main window
- Step 8: Adding a pop-up window
- Step 9: Storing the drawing in a file
- Step 10: Popping up a help window

If you didn't change the default installation directories, you'll find source code in the EXAMPLES\STEPS subdirectory of the main

OWL directory. The files are named STEP1.CPP, STEP2.CPP, and so on, corresponding to the steps in the tutorial.

Figure 2.1 shows the application you will have created at the end of this tutorial.

Figure 2.1
A complete ObjectWindows application



Building an ObjectWindows application: Preliminaries

Before you begin, there are several things you should be aware of when building an ObjectWindows application. First, you need to specify the correct include directories so the compiler can find all relevant files. Second, you need to specify the correct library directories so the linker can find all the libraries your application needs. Finally, you need to create a resource file that contains the appropriate ObjectWindows resources and then bind that resource file to your application.

Container class library

The ObjectWindows class library depends on the container class library in the CLASSLIB subdirectory. All ObjectWindows objects share **Object** (defined in the container class library) as their base. Some of the other classes in the container class library are also used by ObjectWindows classes. You can use any of the container classes in your ObjectWindows programs.

Directories

In addition to the standard header files your application might need, like stdio.h and windows.h, you need to tell the various compiler tools where to locate the ObjectWindows header files like owl.h. To write an ObjectWindows program, you will have to include at least owl.h. More complicated applications will need to include other header files. For example, a program that uses the **TButton** class must include button.h. ObjectWindows applications and ObjectWindows itself rely on the container class library, so you must also include the directory path of the container class library include directory.

All applications need to access the standard run-time libraries, plus any libraries you've created or purchased from third parties. An ObjectWindows application, in addition, needs to have access to the container class libraries and standard ObjectWindows libraries.

The following table summarizes the default directories used by the installation utility. If you told the installation utility to use different directories, you'll need to change the supplied makefiles and project files to reflect your actual setup.

Table 2.1
Default directories

Type	Include directory	Library directory
ObjectWindows	OWL\INCLUDE	OWL\LIB
Container class library	CLASSLIB\INCLUDE	CLASSLIB\LIB
C++	INCLUDE	LIB

Specifying the correct library

There are four different ObjectWindows libraries and container class libraries—one for each of the small, medium, compact, and large memory models. These libraries are located in the library directories listed in the following table and should be used if you want to statically link ObjectWindows and the container class library:

Table 2.2
Libraries for each memory model

Memory model	ObjectWindows library	Container class library
Small	OWLWS.LIB	TCLASSS.LIB
Medium	OWLWM.LIB	TCLASSM.LIB
Compact	OWLWC.LIB	TCLASSC.LIB
Large	OWLWL.LIB	TCLASSL.LIB

ObjectWindows applications that use DLLs

Dynamic link libraries (DLLs) and import libraries for ObjectWindows, the container class library, and the run-time library are available as specified in the following table. If you want your application to use these DLLs, you must adhere to the following restrictions:

*Borland C++ command-line compiler users can also use the command line options **-ml** and **-WS**.*

- Compile your application using the large memory model and the smart callbacks option.
- Link with the import libraries as specified in table 2.3.

*Borland C++ users can also use the **-D** command-line option to define _CLASSDLL.*

- Define the macro _CLASSDLL. You can do this with a `#define _CLASSDLL` directive before the `#include <owl.h>` directive in your ObjectWindows application, or by typing `_CLASSDLL` into the Options | Compiler | Code generation | Defines text box.

- If you use ObjectWindows as a DLL, you must also use the container class library and the run-time library as DLLs.

Table 2.3
DLLs and import libraries

Type	DLL	Import library
ObjectWindows	OWL.DLL	OWL.LIB
Container class library	TCLASS.DLL	TCLASDLL.LIB
Run-time library	BC30RTL.DLL	CRTLDLL.LIB

Creating the resource file

An ObjectWindows resource file is similar to any other Windows resource file with the possible inclusion of ObjectWindows-specific include files. Several resource files (with the .DLG or .RC extensions) are supplied with ObjectWindows. Four of the ObjectWindows classes rely on these resource files to supply the resources used in the classes.

Table 2.4 ObjectWindows classes that require resource files

Classes	Resource file(s) needed
TInputDialog	INPUTDIA.DLG
TFileDialog	FILEDIAL.DLG
TEditWindow	STDWNDS.DLG, EDITACC.RC, EDITMENU.RC
TFileWindow	STDWNDS.DLG, FILEACC.RC, FILEMENU.RC

If your ObjectWindows application uses any of these classes, make sure that the resource file for your application includes the appropriate ObjectWindows resource file. A sample resource file for an application that uses **TInputDialog** and **TFileDialog** might look like this:

```
#include <windows.h>
#include <owlrc.h>
rcinclude inputdia.dlg
rcinclude filedial.dlg
⋮
```

For Borland C++ users

Using the Borland C++ IDE will be much easier, since it automatically informs RC of the include directories you set in the Options | Directories | Include dialog box.

Other resources your application needed would follow the include directives previously discussed. The ObjectWindows-supplied resource files and owlrc.h are located in the ObjectWindows include directory; this directory must be specified as a command-line option to RC. Since windows.h is in the C++ include directory, that directory must also be specified as a command-line option to RC. Use the **–i** command-line option to RC to specify include directories. You can specify multiple directories by repeating the **–i** option.

Note that you can also use the Resource Workshop, as explained in the next section.

For Turbo C++ for Windows users

You can use the Resource Workshop to create resource files for your ObjectWindows applications. Simply use the Resource Workshop's File | Add to project menu option to add any ObjectWindows resource files to your application's resource file. Then

tell the Resource Workshop to create a .RES resource file that you can add to your Turbo C++ project.

Building an ObjectWindows application: The specifics

Now that the preliminaries of building an ObjectWindows application are out of the way, we can turn to the specifics of building an ObjectWindows application. There are three ways to build an ObjectWindows application. The first is to use the IDE's Project Manager. The second and third options are available to Borland C++ users, as they deal with makefiles and command line tools, respectively.

Using the IDE to build an ObjectWindows application

To build an ObjectWindows application in the IDE, you will need to use a *project* file. For more detailed steps on using projects, see the IDE documentation. The general steps involved in building an ObjectWindows application project follow:

- You should select the precompiled headers option to speed up the compilation of the many (and often large) header files required.
- Open a project file.
- Select Options | Application | Windows EXE application type.

- If you use the ObjectWindows DLL, the container class library DLL, the run-time library DLL, or a user DLL with your application, you must select the smart callbacks option and the large memory model and define the macro _CLASSDLL.
- In the Options | Linker | Libraries dialog box, select Static or Dynamic for the ObjectWindows, container class library, and run-time libraries. All libraries must be linked the same way—either all statically or all dynamically.
- Specify the include and library directories. The default directories are listed in Table 2.3.

Hint: Don't add .h header files.

- Add your application's source files (those with extensions of .CPP, .C, and so on) to the project.

Hint: Don't add .DLG dialog resource files.

- Add your application's .RC or .RES resource file to the project. See the earlier section on resource files for more information. In the examples that follow, you'll need to add the STEPS.RC or STEPS.RES resource file.

- Build your application. The IDE Project Manager examines your project to find whether it needs to compile your application's source files; the linker will then link the intermediate .OBJ object files and the libraries; finally the resources will be appended to the .EXE or .DLL file.

Using the Borland C++ command-line tools to build ObjectWindows applications

See page 21 on include and library directories for more information.

If you want to use the Borland C++ command-line compiler to compile and link your ObjectWindows application, it is probably easiest to edit your configuration file (or make one if none exists) to specify your include and library directories. Use the **-I** and **-L** directives in your configuration file to specify the appropriate directories.

To compile and link your application, use a command line similar to the following:

```
BCC -WE myprog.cpp owlws.lib tclasss.lib
```

This command first compiles MYPROG.CPP and then links it with OWLWS.LIB, and TCLASSS.LIB

If you use the ObjectWindows DLL, the container class library DLL, the run-time library DLL, or an ObjectWindows user DLL with your application, you must compile with the **-WS** (smart callbacks) command line option. Note that you must build your application in large memory model (**-ml**) to use ObjectWindows DLLs. You must also define the _CLASSDLL macro. For example,

```
BCC -WS -ml -D_CLASSDLL myprog.cpp owl.lib tclasdll.lib crtldll.lib
```

compiles MYPROG.CPP then links it with the OWL.LIB, TCLASDLL.LIB, and CRTLDLL.LIB import libraries.

You can then compile your application's resource files with a command-line like:

```
RC -i c:\borlandc\owl\include -i c:\borlandc\include myprog.rc
myprog.exe
```

RC compiles the .RC resource file to a binary .RES resource file and then appends the .RES file to your application's .EXE file. The result is an executable ObjectWindows application.

Step 1: A simple Windows application

You'll start your application development by writing a bare-bones ObjectWindows application, called the steps application. This program, found in STEP1.CPP, can serve as the starting point for all programs you write in ObjectWindows. In Step 1 the steps application will instantiate and create the application's main window.

Application requirements

All Windows programs have a main window that appears when the user starts the program. The user quits the application by closing the main window. In an ObjectWindows application, the main window is usually a *window object*. This object is *owned* by the *application object*, which creates and displays the main window, processes Windows messages, and terminates the application. The application object acts as an object-oriented surrogate for the application itself. In the same way, ObjectWindows provides window, dialog, and other classes to hide the details of Windows programming.

Every ObjectWindows program must define a new application class that descends from the supplied class, **TApplication**. In **WinMain** (the entry point of a Windows program), an instance of this derived class (the *application object*) is constructed. By convention, types (classes and instances of classes) are usually prefixed by the letter *T* and pointers to types by the letters *PT*. In the steps application, this class is called **TMyApp**.

Here is the main function of the steps application:

This code fragment is from STEP1.CPP.

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  TMyApp MyApp("Sample ObjectWindows Program", hInstance,
               hPrevInstance, lpCmdLine, nCmdShow);
  MyApp.Run();
  return MyApp.Status;
}
```

In the first statement, **WinMain** constructs the *application object* of the steps application. "Sample ObjectWindows Program" is passed to the constructor to become the value of its *Name* data member. Also, the other parameters passed to **WinMain** are

supplied to the constructor and stored as data members in the application object. **MyApp.Run** is then called to set the ObjectWindows application in motion. The final status (the application object's *Status* data member) is returned in the last statement.

Defining the application class

Your application must derive a new class from the standard ObjectWindows class **TApplication** (or some class derived from **TApplication**). This new class should redefine at least one virtual member function, **InitMainWindow**. **InitMainWindow** constructs a main window object during initialization of an ObjectWindows application. Here's the definition of the class **TMyApp**:

This code fragment is from STEP1.CPP.

```
class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
         lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
         hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};
```

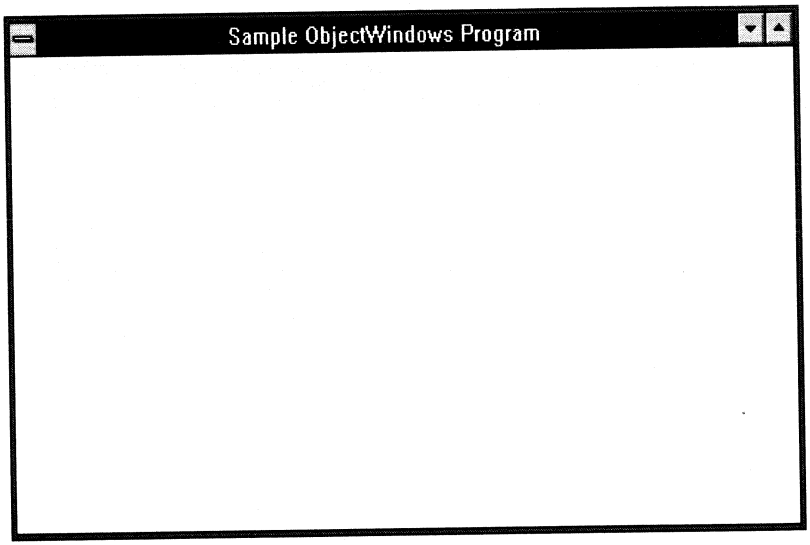
This main window object is stored in the application object's *MainWindow* data member. The application object is said to *own* the main window object, but the two are not related hierarchically. This ownership relationship is called an *instance linkage*. The definition of **TMyApp::InitMainWindow** follows:

This code fragment is from STEP1.CPP.

```
void TMyApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, Name);
}
```

The main window object of **TMyApp** is constructed, above, as an instance of the **TWindow** class. The supplied parameter NULL indicates the window is to be the main window of your application; the second parameter is the window's title. We pass the *Name* data member of the application object which has already been set to "Sample ObjectWindows program." Your main window object will normally be constructed as an instance of a class you derive from ObjectWindows classes. In Step 2, you'll replace the generic window constructed here with a more interesting window of a derived class. Figure 2.2 shows the appearance of the steps application.

Figure 2.2
Your first ObjectWindows program, the steps application



At this point, the steps application displays a blank window that can be moved, resized, maximized, minimized, and closed. Here is a full listing of the steps application up to this point:

This is STEP1.CPP.

```
#include <owl.h>

class TMyApp : public TApplication
{
public:
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
         lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
         hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

void TMyApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, Name);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TMyApp MyApp("Sample ObjectWindows Program", hInstance,
               hPrevInstance, lpCmdLine, nCmdShow);
  MyApp.Run();
  return MyApp.Status;
}
```

Step 2: The main window class

The program you created in Step 1 consists of two objects: an application object and a window object. The application object, **MyApp**, is an instance of class **TMyApp**, a class you derived from **TApplication**. The window object pointer, held in the *MainWindow* data member of the application object, points to an instance of **TWindow**, a generic ObjectWindows window. You'll normally define your own window class for the main window, incorporating application-specific behavior. In this section, you will bring the main window object to life by defining a specialized window class, derived from **TWindow**.

What is a window object?

In Step 1 you saw that an application object encapsulates the standard behaviors of a Windows application, including construction of the main window. Class **TApplication** provides the fundamental behaviors of your ObjectWindows applications.

Similarly, a window object encapsulates a window's required behavior, including the ability to

- display and resize itself
- respond to events
- manage child windows
- close in an orderly fashion

The ObjectWindows window classes provide this behavior.

To make your programs useful and interesting, you will have to create new window classes derived from the supplied ObjectWindows classes. The new classes will inherit member functions and data members, and add some of their own. Overall, the object-oriented approach will save you from constantly "re-inventing the window."

Handles

All window objects have at least three data members: *HWindow*, *Parent*, and its child list. *HWindow* holds the handle to the window. A handle associates an interface object, such as a window, dialog box, or control object, with its corresponding interface element. It's a lot like a claim number at a coat check. Just as you present a claim check to identify your coat, you present your handle to identify your window.

In most of your work with window objects, you will not have to directly manipulate the window handle. It is needed, however, when calling Windows functions directly. For example, to bring up a message box, you call the Windows API **MessageBox** function. **MessageBox** requires the handle of the message box's parent window. The **MessageBox** call below appears as it might in a member function of a derived window class. For information about Windows API functions, see the online Help.

F1
Help

```
MessageBox(HWindow, "Do you want to save?", "Drawing has changed",
           MB_YESNO | MB_ICONQUESTION);
```

Parents and children

Parent window relationships are described in Step 8 starting on page 62.

Most windows do not exist independently of others: They are linked together and act in concert. For example, when you terminate an application, all the windows it is responsible for must be destroyed. In general, Windows links windows as parents and children. A parent window is responsible for its children. ObjectWindows provides data members for each window object to keep track of its parent and child windows.

The *Parent* data member holds the window's parent window object. This is not a parent as in a derived class's base class, but more like an owner window.

The third window data member is its child list, which holds a list of pointers to the window's *child windows*, if any. You will add child windows to your program in Step 8.

Creating a main window object

Now that you have some idea of what a window object holds, you can define a new window class, derived from class **TWindow**, to serve as a main window for the steps application. First, update the class declarations to specify the new class, **TMyWindow**.

This code fragment is from STEP2.CPP.

```
_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
public:
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
    TWindow(AParent, ATitle) {};
};
```

Notice that the constructor for the new derived class does not define any new behavior and merely invokes the base class's (**TWindow**) constructor.

Next, update **TMyApp::InitMainWindow** so it constructs a **TMyWindow**, rather than a **TWindow**, as its main window.

This code fragment is from STEP2.CPP.

```
void TMyApp::InitMainWindow()
{
  MainWindow = new TMyWindow(NULL, Name);
}
```

Declaring the new class and instantiating it in **InitMainWindow** is all that's required to define a new class for **TMyApp**'s main window. The application object calls member functions of the window object to create the window interface element and to display it on the screen. You don't need to worry about it.

However, **TMyWindow** defines no new behaviors beyond those inherited from **TWindow** and **TWindowsObject**. In other words, it doesn't make the steps application any more interesting. In the next section, you will add some interesting behavior.

Responding to messages

The quickest way to make a window object useful is to teach it how to respond to Windows messages. For example, when the user clicks the left mouse button in the main window of the steps application, the corresponding window object receives a WM_LBUTTONDOWN message from Windows. This tells the window object that the user clicked the mouse in it. It also passes the coordinates of the point where the user clicked. (This information is used in Step 6 of this tutorial.) Similarly, when the user clicks the right mouse button, the main window object receives the WM_RBUTTONDOWN message from Windows.

The next step is to teach the main window, an instance of **TMyWindow**, how to respond to these messages and do something useful. To intercept and respond to Windows messages, you must define a member function for your window class for each type of incoming message you want to respond to. These member functions are called message response member functions. You'll define a message response member function for each Windows message you want to respond to. (If you don't define a response for a particular message, ObjectWindows will perform its default processing.) To mark a member function as a response member

function, add a dispatch index to its declaration, as discussed on page 9. Note that the name of the function isn't important, but its dispatch index is. The convention we use for message response function names is to mix the capitalization of and remove the underscores from the Windows message constant.

This code fragment is from STEP2.CPP.

```
class TMyWindow : public TWindow {
public:
:
virtual void WMLButtonDown(RTMessage Msg) =
  [WM_FIRST + WM_LBUTTONDOWN];
:
};
```

Msg is a reference to a **TMessage** structure (shortened using the ObjectWindows type **RTMessage**) that holds information about the event that occurred. (You'll learn more about this structure in Step 3.) All message response member functions are passed an **RTMessage** and are return type **void**.

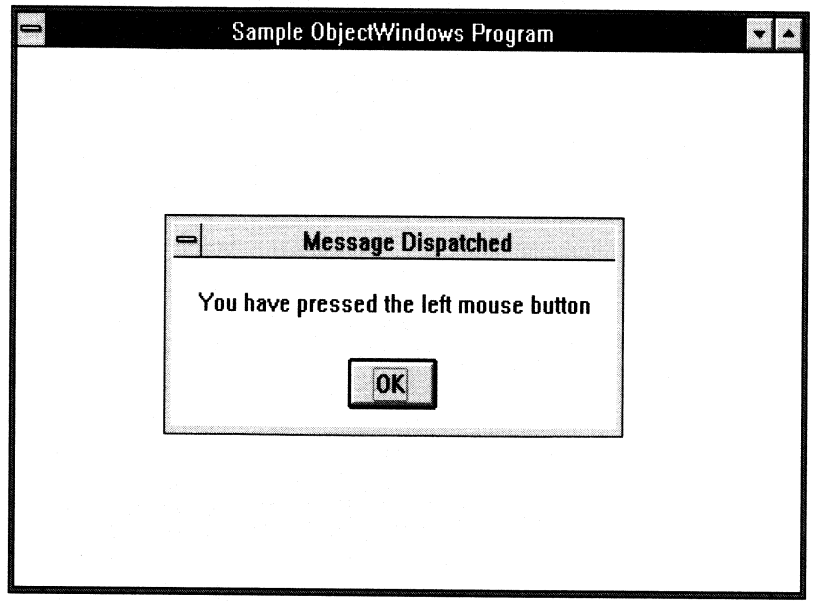
For now, just define response member functions that put up message boxes announcing that the mouse buttons have been pressed. Later, you will add more useful responses. Here is the definition for the left button response member function:

This code fragment is from STEP2.CPP.

```
void TMyWindow::WMLButtonDown(RTMessage)
{
  MessageBox(HWindow, "You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}
```

Figure 2.3
The steps application responding to a user event

The full source code for this step is listed at the end of this chapter.



Terminating an application

The program created here closes when the user double-clicks the Control-menu box of its main window, the small square in its upper left corner. The window and the application close immediately. This behavior is fine for simple programs, but may not be for some others.

For example, have you ever quit an application without saving your work? A good application always asks if the user wants to save work when it has not yet been saved. You can easily add this behavior to your ObjectWindows applications. Take the steps application and add the ability to double-check the user's request to quit.

When the user tries to close your ObjectWindows application, the virtual **CanClose** member function of your application class is invoked. **CanClose** is a **BOOL** function that indicates whether it is OK (TRUE) to close the application. As a default, the **CanClose** member function inherited from **TApplication** calls the **CanClose** member function of the main window object. In most cases, it is the main window object that decides if it is OK to close.

Your main window class, **TMyWindow**, inherits a **CanClose** member function from **TWindowsObject** that calls the **CanClose**

member functions of each of its child windows, if any. If there are no child windows (as in this case), **CanClose** simply returns TRUE. To modify an application's closing behavior, you'll redefine a **CanClose** member function for your main window class:

This code fragment is from STEP2.CPP.

```
BOOL TMyWindow::CanClose()
{
  return MessageBox(HWindow, "Do you want to save?",
    "Drawing has changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}
```

Now, when users try to close the steps application, they are presented with a message box that asks, "Do you want to save?" Clicking the Yes button returns FALSE and prevents the main window and application from closing. Clicking the No button returns TRUE and terminates the application. Figure 2.4 shows the modified steps application.

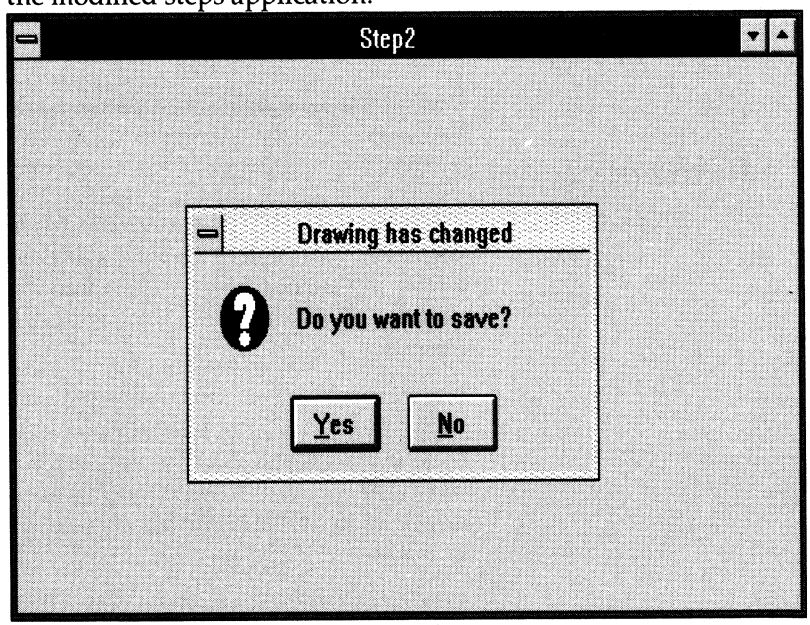


Figure 2.4
The steps application with refined closing behavior

Here is the full source code to the steps application thus far:

This is STEP2.CPP.

```
#include <owl.h>

class TMyApp : public TApplication
{
public:
```

```
  TMyApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
         lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
         hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
public:
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle) : TWindow(AParent,
           ATitle) {};
  virtual BOOL CanClose();
  virtual void WMLButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_LBUTTONDOWN];
  virtual void WMRButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_RBUTTONDOWN];
};

BOOL TMyWindow::CanClose()
{
  return MessageBox(HWindow, "Do you want to save?", "Drawing has
                    changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}

void TMyWindow::WMLButtonDown(RTMessage)
{
  MessageBox(HWindow, "You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}

void TMyWindow::WMRButtonDown(RTMessage)
{
  MessageBox(HWindow, "You have pressed the right mouse button",
             "Message Dispatched", MB_OK);
}

void TMyApp::InitMainWindow()
{
  MainWindow = new TMyWindow(NULL, Name);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TMyApp MyApp("Sample ObjectWindows Program", hInstance,
               hPrevInstance, lpCmdLine, nCmdShow);
  MyApp.Run();
  return MyApp.Status;
}
```

C H A P T E R

3

Filling in the window

In this chapter, you'll take the steps application, which at present is just a shell of a program, and transform it into a useful, interactive graphics application. First you'll draw text on the steps application's main window. Then you'll transform the steps application into a small graphics application that lets you draw lines on the main window. After that, you'll refine the drawing program to redraw its graphics, change the thickness of the lines, and finally, save its graphics into a file for reloading at a later time.

What in the world is a display context?

You can think of a display context as representing the drawing surface of a window. A display context is required by Windows for drawing text or graphics in a particular window. A handle to a display context, its identifier, must be passed to Windows Graphics Device Interface (GDI) "drawing" functions.

To draw on a window, you must first obtain a display context. Just before drawing to the screen, call the Windows function **GetDC** from within one of the window object's member functions:

```
TheDC = GetDC(HWindow);
```

GetDC returns the display context's handle (of a Windows-defined **HDC** type), which you can now pass to Windows GDI functions. Here are a few example GDI calls:

```
LineTo(TheDC, 30, 45);
Rectangle(TheDC, 100, 100, 200, 200);
TextOut(TheDC, 20, 20, "Sample text", 11);
```

After drawing text or graphics, you *must* release the display context. Make sure you release any display context you get as soon as you're done drawing.

```
ReleaseDC(HWindow, TheDC);
```

If you do not release all obtained display contexts, you will soon run out of them (the entire Windows environment has a limit of five), and your application will fail, usually causing your computer to hang. If your application fails the third or fourth time you draw something, unreleased display contexts are the first thing you should check.

The display context has some important functions. First, it ensures that you do not draw text and graphics outside the boundaries of your window. Text and graphics that you output by calling GDI functions are *clipped* to the boundaries of the window for which the display context was obtained.

The display context also maintains a set of drawing tools (a pen, brush, font, and palette) used when GDI functions are called. In the previous examples, the pen selected in the display context will be used to draw the line when **LineTo** is called; the line drawn will have the characteristics of the selected pen (including color and width). Similarly, the brush selected in the display context will be used to fill the specified rectangular area of the display when **Rectangle** is called. The selected font will be used to display "Sample Text" when **TextOut** is called.

You may select new tools into a display context in order to change the manner in which your output is displayed. For example, you may wish to use a brush with a pattern, a pen of a different color, or a different style font. Selection of a new pen is demonstrated in Step 5.

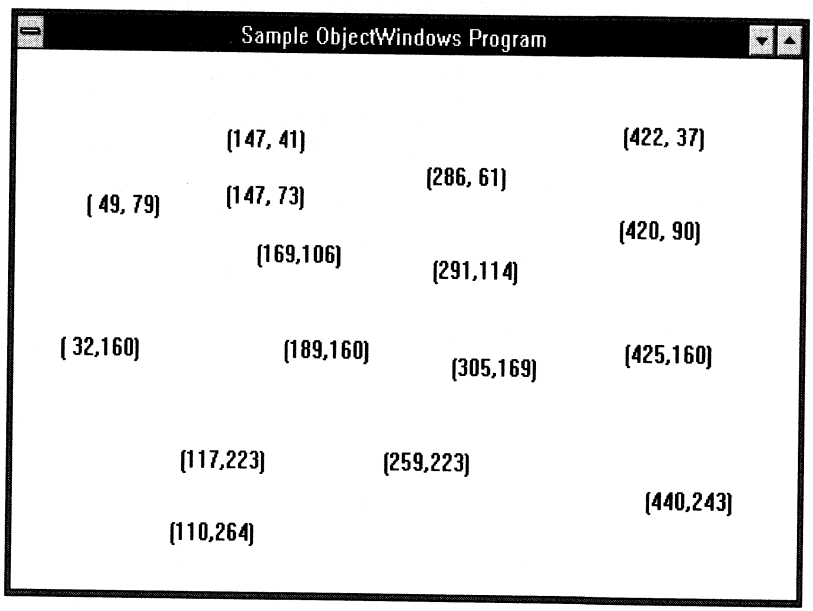
Step 3: Drawing text in a window

To draw text in the main window of the steps application, follow the display context drawing model introduced in the preceding section: First obtain a display context, then perform the display operations, and finally release the display context. To make things

interesting, draw the text in response to the left mouse button clicks you originally intercepted in Step 2. Instead of bringing up a message box, this time you'll respond by drawing text that shows the coordinates of the point where you clicked on the window.

This example also helps you understand the Windows coordinate system. For example, "(20,30)" is drawn when the mouse is clicked at the point 20 pixels to the right of, and 30 pixels down from, the top left corner of the window's drawing surface (its *client area*). The text will be drawn at the point the mouse was clicked.

Figure 3.1
The steps application drawing text where the user clicks



Remember that a left button click event generates a WM_LBUTTONDOWN message, which you intercept with a message response member function called **WMLButtonDown**.

Message structure

In Step 2, you read that the *Msg* argument, a **TMessage** structure reference (shortened to type **RTMessage**) supplied to a message response member function, contains information about the event that occurred. This data is contained in the *LParam* and *WParam* members (of type **LONG** and **WORD**, respectively) of the structure.

Msg.LParam often contains two pieces of information, one in its high- and one in its low-order word. *LParam* is declared as a member of a union along with an **LP** structure, which declares *Hi* and *Lo* members of type **WORD**. You can therefore use *Msg.LP.Hi* and *Msg.LP.Lo*. to refer to the high- and low-order words of *Msg.LParam*. *WParam* is similarly declared as a member of a union along with a **WP** structure. However, Windows rarely uses the **WP** structure's *Hi* and *Lo* members.

In the case of **WMLButtonDown**, *Msg.LP.Lo* holds the x-coordinate of the clicked point, and *Msg.LP.Hi* holds the y-coordinate. To rewrite **WMLButtonDown** to draw the coordinates of the clicked point, you'll need to convert *Msg.LP.Lo* and *Msg.LP.Hi* into text, and add punctuation. **sprintf** was used in this example to build the text string to be drawn.

Once you've built the final string, you can draw it at the point that was clicked by calling the Windows **TextOut** function, passing the display context, the coordinates of the point, (*Msg.LP.Lo* and *Msg.LP.Hi*), the string, *S*, and the string length, **strlen**(*S*). Also, be sure to obtain the display context before drawing and release it afterward.

This code fragment is from STEP3.CPP.

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  HDC DC;
  char S[10];

  sprintf(S, "(%d,%d)", Msg.LP.Lo, Msg.LP.Hi);
  DC = GetDC(HWindow);
  TextOut(DC, Msg.LP.Lo, Msg.LP.Hi, S, strlen(S));
  ReleaseDC(HWindow, DC);
}
```

Clearing the screen

One more function you can add to the text drawing application is clearing the screen. Notice that once you resize the window, or cover and reveal it, the drawn text is erased. However, you might want to force screen clearing in response to a menu choice or some other user action, such as a mouse click.

You'll clear the window in response to a right mouse button click. To implement this, redefine the **WMRButtonDown** member function to call the Windows **InvalidateRect** function, whose arguments specify that the whole window be repainted. Since

your window doesn't yet know how to repaint itself, it just clears its client area:

This code fragment is from STEP3.CPP.

```
void TMyWindow::WMRButtonDown(RTMessage)
{
  InvalidateRect(HWindow, NULL, TRUE);
}
```

You can see the current source code in the file STEP3.CPP.

Step 4: Drawing lines in a window

Now that you've seen the drawing model (in which you obtain a display context, draw, then release the display context), you can use it in a more complete, interactive graphics application. In this section, you'll build a simple painting program that lets the user draw on the main window.

You will take the following steps:

1. Respond to left mouse button clicks and drags by connecting the dots along the way, resulting in drawn lines.
2. Respond to right mouse button clicks by bringing up an input dialog, allowing the user to change the line thickness.
3. Make the window redraw its contents by storing the points and redrawing them in response to a **Paint** message.

To accomplish these things, you'll first study the Windows dragging model, then implement a simple graphics drawing program.

The dragging model

It will be helpful at this point to review the Windows mouse event model. You have already seen that by using the DDVT support discussed in Chapter 2, a left mouse button click results in a WM_LBUTTONDOWN message and a **WMLButtonDown** member function call. Earlier in this tutorial, you responded to left mouse button clicks by bringing up message boxes and by drawing text on the screen. You also saw that a right mouse button click results in a WM_RBUTTONDOWN message and a **WMRButtonDown** member function call. You responded to right mouse button clicks by clearing the screen. But these responses

only cover the initial clicks of a mouse button. Many interactive Windows programs require you to click and drag the mouse around on the screen to draw lines or rectangles, or to place graphics in particular locations. For your graphics drawing program, you want to capture the dragging events and respond by drawing lines.

You do this by responding to a few more messages. WM_MOUSEMOVE is received when the user drags the mouse to a new point in a window, and WM_LBUTTONUP is received when the user releases the left mouse button. Typically, a window will receive one WM_LBUTTONDOWN message, followed by a series of WM_MOUSEMOVE messages, followed by one WM_LBUTTONUP message. A typical graphical Windows program will respond to WM_LBUTTONDOWN by initiating the drawing process (obtaining a display context, among other things). It will respond to WM_MOUSEMOVE by drawing or moving graphics, and it will respond to WM_LBUTTONUP by terminating the drawing process (releasing the display context).

The following table summarizes the most common mouse event messages.

Table 3.1
Common mouse event messages

Message	Event
WM_LBUTTONDOWN	The user clicks the left mouse button.
WM_RBUTTONDOWN	The user clicks the right mouse button.
WM_MOUSEMOVE	The user drags the mouse.
WM_LBUTTONUP	The user releases the left mouse button.
WM_RBUTTONUP	The user releases the right mouse button.
WM_LBUTTONDBLCLK	The user double-clicks the left mouse button.
WM_RBUTTONDBLCLK	The user double-clicks the right mouse button.

Responding to drag messages

The following table summarizes how you respond to drag messages to create your line-drawing program:

Table 3.2
Messages used in Step 4

Message	Response
WM_LBUTTONDOWN	Clear the screen. Then obtain a display context and store it in *DragDC*. Position the drawing pen at the point clicked.
WM_MOUSEMOVE	Draw a line from the previous point to the current point if the mouse button is still down.
WM_LBUTTONUP	Release *DragDC*.

As stated previously, WM_LBUTTONDOWN is always followed by WM_LBUTTONUP, with or without WM_MOUSEMOVE messages in between. You must obtain a display context when your window receives a WM_LBUTTONDOWN message and release it on a WM_LBUTTONUP. You'll use a single display context for all the drawing that takes place while the user clicks the left mouse button.

Pairing the obtaining of a display context with its release is critical to the proper functioning of your graphics programs. However, you can also add one more safety measure. Define a new **BOOL** data member for **TMyWindow**, the main window class, called *ButtonDown*. **WMLButtonDown** will set it to TRUE and **WML-ButtonUp** sets it to FALSE. Then you can check the value of *ButtonDown* before obtaining and releasing the display context.

Here are the three mouse drag member functions:

This code fragment is from STEP4.CPP.

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    ButtonDown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
  }
}

void TMyWindow::WMMouseMove(RTMessage Msg)
{
  if ( ButtonDown )
    LineTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
}

void TMyWindow::WMLButtonUp(RTMessage)
{
  if ( ButtonDown )
  {
    ButtonDown = FALSE;
    ReleaseCapture();
    ReleaseDC(HWindow, DragDC);
  }
}
```

MoveTo and **LineTo** are graphics functions in the Windows API that move the current drawing position and draw a line to the current position, respectively. They require a handle to the

display context that **TMyWindow** stores in its *DragDC* data member. (Remember that you are not drawing directly on the window, but on its display context.)

If the mouse is dragged outside of a **TMyWindow**, WM_MOUSEMOVE messages will still be sent to the **TMyWindow**, even when the mouse is positioned over an adjacent window. This happens as a result of the **TMyWindow** capturing mouse input via a call to **SetCapture**. (Mouse input is released via a call to **ReleaseCapture**.)

Be sure to update the object definition for **TMyWindow** with member function headers for **WMMouseMove** and **WMLButtonUp**:

This brings the code up to STEP4.CPP.

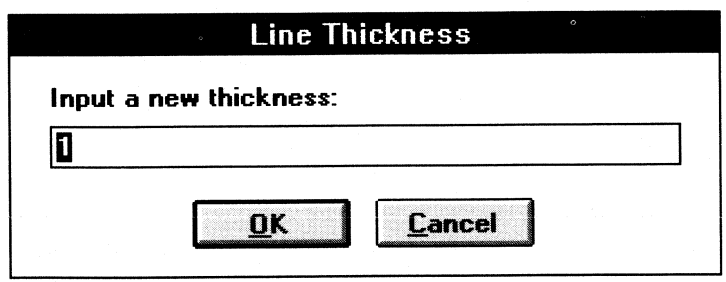
```
virtual void WMLButtonDown(RTMessage Msg)
  = [WM_FIRST + WM_LBUTTONDOWN];
virtual void WMLButtonUp(RTMessage Msg)
  = [WM_FIRST + WM_LBUTTONUP];
```

Step 5: Changing the line thickness

At this point, you can draw only thin lines. But drawing programs traditionally let you change the thickness of the lines that you draw. When you do this, you're not really changing the thickness of the lines, but the thickness of the pen you use to draw the lines. Pens, as well as brushes, fonts, and palettes, are drawing tools embodied by a display context. In this step, you'll learn how to set new tools in a display context while giving the steps application the ability to set a new line thickness.

You'll also use an input dialog (of class **TInputDialog**) to provide a mechanism for the user to change the line thickness. Figure 3.2 shows the input dialog in use.

Figure 3.2
Changing line thickness



Selecting a new pen

To change the thickness of the drawn lines, you need to first understand a little more about Windows graphics and display contexts in particular.

The drawing tools used to produce graphics and text in a window are pens, brushes, and fonts. The descriptions of these drawing *elements* are stored in Windows memory, as are interface elements. Both types of elements are therefore identified using handles. However, drawing elements are not represented by Object-Windows objects. As a result, your program has full responsibility for creating and deleting the drawing tools it uses.

Forgetting to delete your tools can result in significant loss of Windows memory.

Think of a drawing tool as a painter's paintbrush and a display context as the canvas. Once a painter has all the tools (paintbrushes) and a display context (the canvas), then the painter selects the drawing tools to be used. Similarly, a Windows program must select drawing tools *into* a display context. So, how is it that you can draw text and lines in your windows without selecting any drawing tools? All display contexts come with a set of default tools: a thin black pen, a solid black brush, and a system font. In this step, you will select a thicker pen for drawing in the window.

The first step is to add support for input dialogs to the steps application. The header file that defines input dialog support is inputdia.h. Add this to your .CPP source file. Input dialogs assume that you have defined a dialog resource in a .RC file. The input dialog resource is defined in INPUTDIA.DLG. Include this file in your .RC file. STEPS.RC contains all the resource definitions needed for the steps application. At this point, all you'd need in your .RC file is

```
#include "windows.h"
#include "owlrc.h"
rcinclude inputdia.dlg
```

Next, add a data member to **TMyWindow** to store a handle to the pen tool you will use to draw graphics. In this program, limit yourself to drawing and displaying lines in only one thickness at a time. The pen corresponding to this thickness will be stored in the new **TMyWindow** data member called *ThePen*. You'll also write

a member function, **SetPenSize**, to create the new pen tool and delete the old pen tool. Your **TMyWindow** class declaration now looks like this:

This code fragment is from STEP5.CPP.

```
_CLASSDEF(TMyWindow)
class TMyWindow : public TWindow
{
public:
  HDC DragDC;
  BOOL ButtonDown;
  HPEN ThePen;
  int PenSize;
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle);
  ~TMyWindow();
  virtual BOOL CanClose();
  void SetPenSize(int NewSize);
  virtual void WMLButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_LBUTTONDOWN];
  virtual void WMLButtonUp(RTMessage Msg)
    = [WM_FIRST + WM_LBUTTONUP];
  virtual void WMMouseMove(RTMessage Msg)
    = [WM_FIRST + WM_MOUSEMOVE];
  virtual void WMRButtonDown(RTMessage Msg)
    = [WM_FIRST + WM_RBUTTONDOWN];
};
```

To initialize these new data members, modify the constructor to set up the pen, and define the destructor to destroy the pen.

This code fragment is from STEP5.CPP.

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
}

TMyWindow::~TMyWindow()
{
  DeleteObject(ThePen);
}
```

Now alter the **WMLButtonDown** member function to select the current pen (*ThePen*) into the newly obtained display context.

This code fragment is from STEP5.CPP.

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    ButtonDown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
  }
}
```

Like MoveTo and MessageBox, SelectObject is a Windows API function.

Changing the pen

The member function previously described selects the already-created pen into the display context. However, to create a new pen, you need to write the following **SetPenSize** member function:

This code fragment is from STEP5.CPP.

```
void TMyWindow::SetPenSize(int NewSize)
{
  DeleteObject(ThePen);
  ThePen = CreatePen(PS_SOLID, NewSize, 0);
  PenSize = NewSize;
}
```

Calling the Windows function **CreatePen** is one way to create a Windows pen tool with the specified thickness: You store a handle to the pen in *ThePen*. Deleting the previous pen is a very important step: If you don't delete it, you will slowly and irretrievably use up Windows memory.

Running the input dialog

Clicking the right mouse button is a convenient way to bring up the option to change the line thickness. Let's redefine the **WMRButtonDown** member function to bring up an input dialog box, one of ObjectWindows' stock dialogs. An input dialog box serves as a simple dialog box that gets one line of text input from the user.

Once the input dialog object has been constructed, you can run it as a modal dialog by calling **TModule::ExecDialog**. **ExecDialog**, used for dialog objects, is similar to **MakeWindow**, which is used for window objects and dialog objects that are used as modeless dialog boxes. Both will create an interface element only if the

interface object can be validated. **ExecDialog**, however, returns only after the user has closed the dialog by clicking OK or Cancel. If the user clicks OK, *InputText* is filled with the user's input. Since you are asking for a thickness number, you must convert the returned text to a number and pass it in a call to **SetPenSize**. Every time the user chooses a new line thickness, delete the old pen and create a new pen.

The code for the program to this point can be found in STEP5.CPP.

```
void TMyWindow::WMRButtonDown(RTMessage)
{
  char InputText[6];
  int NewPenSize;

  sprintf(InputText, "%d", PenSize);
  if ( GetApplication()->ExecDialog(new TInputDialog(this, "Line
                                    Thickness", "Input a new
                                    thickness:", InputText, sizeof
                                    InputText)) == IDOK )
  {
      NewPenSize = atoi(InputText);
      if ( NewPenSize < 0 )
        NewPenSize = 1;
      SetPenSize(NewPenSize);
  }
}
```

Step 6: Painting graphics

You might be surprised to learn that the graphics and text you draw in a window using Windows functions like **TextOut** and **LineTo** disappear when you resize or uncover the window. Where did they go?

A better question is, "Where were they in the first place?" You never stored the text or lines in any type of variable. Once the graphics data goes to Windows through calls to Windows functions, you can't get it back to redraw it when needed. You've got to store the graphics in some other type of structure to have a window re-display its graphics; an object is well-suited to the task. Objects can store simple or complex graphics, and can be easily maintained as data members of the main window.

The painting model

Windows sends a WM_PAINT message to your application when the display of one of your windows requires updating, or *painting*. For example, a WM_PAINT message is sent when one of your windows is resized or uncovered by the user. ObjectWindows intercepts the WM_PAINT message, and calls the **WMPaint** member function of your window.

TWindowsObject defines a **WMPaint** member function that calls the virtual function **Paint**, performing windows-required setup and cleanup, before and after the call. Redefine **Paint** in classes derived from **TWindow** to paint your window's display.

There is one major difference between drawing graphics in the **Paint** member function and at other times, such as in response to mouse actions. The display context to be used for painting is passed in the *PaintDC* parameter, so your program need not obtain or release it. You will, however, need to reselect your drawing tools into *PaintDC*.

To paint your window's contents, replay the actions that led to the original drawing on *DragDC*, but use *PaintDC* instead. The visual effect will be the same as when they were drawn the first time by the user, much like replaying an audio recording of a concert: Is it live or is it ObjectWindows? But first, you need to store the graphics data your **Paint** member function requires.

Storing graphics as objects

In Step 4, you saw that the line was drawn originally in response to WM_MOUSEMOVE messages. These messages contain the current mouse position as x- and y-coordinates of a point on the window's display. The line was drawn by connecting the points sent along with these messages. The points that were received are the graphics data needed to redraw the line.

We'll use **TPoint** objects, a class derived from **Object**, to represent these points. (You could use a simple structure here, rather than an object, but you'll be storing these **TPoint** objects in one of the container class library's containers, which all require objects derived from **Object**. Furthermore, in future steps you'll be adding member functions to **TPoint**.)

We'll store **TPoint** objects in a **TMyWindow** data member called *Points*. *Points* will be an instance of type **TPointArray** which we will derive from the container class **Array**. The reason we'll use **TPointArray** rather than **Array** is that we'll be adding new functionality to **TPointArray** not available in **Array**. Because **Array** objects (and classes derived from **Array**) know how to grow dynamically, they're well-suited to holding a variable number of points. **Array** objects and the other container classes are documented in your compiler documentation. Here is part of the **TPoint** definition:

```
class TPoint: public Object {
public:
  int X, Y;
  TPoint(int AX, int AY) {X = AX, Y = AY;}
⋮
};
```

Here is part of the **TPointArray** declaration:

This code fragment is from STEP6.CPP.

```
_CLASSDEF(TPointArray)

class TPointArray : public Array
{
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0) :
              Array(upper, lower, aDelta) {};
  void DeleteAll();
⋮
};
```

Now let's look at the updated **TMyWindow** declaration with the addition of the *Points* data member:

This code fragment is from STEP6.CPP.

```
_CLASSDEF(TPointArray)

class TPointArray : public Array
{
⋮
  PPointArray Points;
⋮
};
```

Be sure to construct *Points* in **TMyWindow**'s constructor and **delete** it in **TMyWindow**'s destructor.

This code fragment is from STEP6.CPP.

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
  Points = new TPointArray(50, 0, 50);
}

TMyWindow::~TMyWindow()
{
  delete Points;
  DeleteObject(ThePen);
}
```

You must alter **WMLButtonDown** and **WMMouseMove** to set up *Points*.

This code fragment is from STEP6.CPP.

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
  if ( !ButtonDown )
  {
    ButtonDown = TRUE;
    SetCapture(HWindow);
    DragDC = GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
    Points->add(* (new TPoint(Msg.LP.Lo, Msg.LP.Hi)));
  }
}

void TMyWindow::WMMouseMove(RTMessage Msg)
{
  if ( ButtonDown )
  {
    LineTo(DragDC, Msg.LP.Lo, Msg.LP.Hi);
    Points->add(* (new TPoint(Msg.LP.Lo, Msg.LP.Hi)));
  }
}
```

See the container class library documentation for a description of how to use an ArrayIterator.

WMLButtonUp requires no modification. In **TMyWindow::WMLButtonDown**, the *Points* **Array** is cleared of **TPoint**s before beginning a new line by calling the **TPointArray::DeleteAll** member function:

This code fragment is from STEP6.CPP.

```
void TPointArray::DeleteAll()
{
  RArrayIterator PointIterator = (RArrayIterator)initIterator();

  while ( int(PointIterator) != 0 )
  {
    RObject AnObject = PointIterator++;
    if ( AnObject != NOOBJECT )
      detach(AnObject, 1);  // detach and delete
  }
  delete &PointIterator;
}
```

Redrawing stored graphics

Now that *Points* has been set up, you must write the code to paint a **TMyWindow**, redrawing lines between each of the points. We'll write a **Paint** member function for **TMyWindow** that repeats the actions performed in its **WMLButtonDown** and **WMMouseMove** member functions, using its *Points* data member. Here is the **Paint** member function:

This code fragment is from STEP6.CPP.

```
void TMyWindow::Paint(HDC DC, PAINTSTRUCT&)
{
  RArrayIterator PointIterator =
                (RArrayIterator)(Points->initIterator());
  BOOL First = TRUE;

  SelectObject(DC, ThePen);
  while ( int(PointIterator) != 0 )
  {
    RObject AnObject = PointIterator++;
    if ( AnObject != NOOBJECT )
    {
      if ( First )
      {
        MoveTo(DC, ((PTPoint)(&AnObject))->X,
                   ((PTPoint)(&AnObject))->Y);
        First = FALSE;
      }
      else
        LineTo(DC, ((PTPoint)(&AnObject))->X,
                   ((PTPoint)(&AnObject))->Y);
    }
  }
  delete &PointIterator;
}
```

To review, the main window of the steps application holds a pointer to a **TPointArray** in its *Points* data member. As the user draws a line, the window receives WM_MOUSEMOVE messages containing mouse positions. You must construct **TPoint** objects that contain this positional data, and add each **TPoint** to the *Points* array. Then, when the window requires repainting, you can redraw the line by connecting the **TPoint**s.

C H A P T E R

4

Adding a menu

Most Windows applications have a menu on their main window to provide a variety of selections for the user, such as File | Save, File | Open, and Help. In this chapter, you'll add a standard menu to the steps application.

In a windowing environment, a menu selection is in the same category as a mouse click: They're both user events. Responding to a menu selection is a lot like responding to other user events. This chapter traces the required steps to add a menu to an application:

1. Design the menu as a menu resource.
2. Load the menu resource into the main window object.
3. Define responses to menu selections.
4. Compile the application, producing an executable file.
5. Append the menu resource to the executable file.

Menu resources

Somewhere in an application with a menu, there must be a menu specification that contains the text of the menu selections and the structure of the top-level items and their subitems. Also, many applications provide keyboard shortcuts (called *accelerators*) for menu options. Rather than selecting Edit | Paste using the mouse, a user might be able to press *Shift-Ins* to trigger the same response.

Menus and their accelerators are not part of the program's C++ source code. They're part of a separate specification called a *resource*. Windows stores resources in a compact and efficient way.

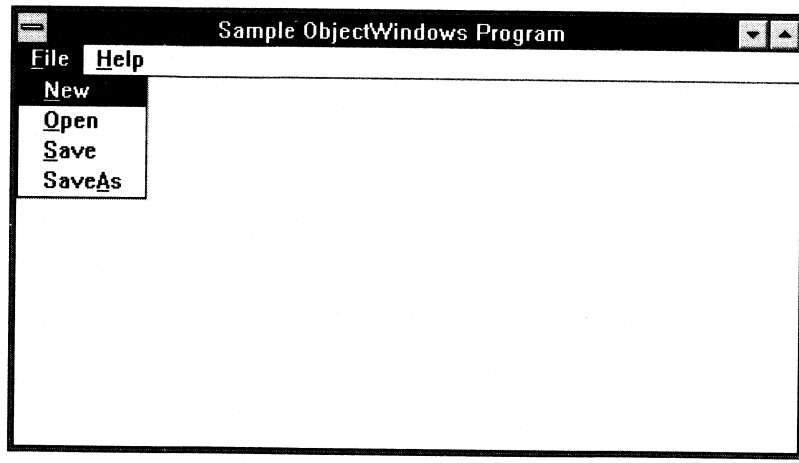
An application accesses its attached resources by specifying the resource ID. This ID is an integer, such as 100, or a string identifier, such as "MyMenu". An application distinguishes one menu selection from another by the menu ID associated with each menu item.

You must use a resource editor or RC, the command-line Resource Compiler, to create a menu resource. STEPS.RC contains the steps application's menu resource in source code format. If you're using the Turbo C++ IDE, you can add STEPS.RC to the steps application's project, so the IDE automatically translates it using RC. Here is STEPS.RC:

```
#include <windows.h>
#include <owlrc.h>
#include "steps.h"

rcinclude inputdia.dlg
rcinclude filedial.dlg

COMMANDS MENU LOADONCALL MOVEABLE PURE DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MenuItem  "&New", CM_FILENEW
    MenuItem  "&Open", CM_FILEOPEN
    MenuItem  "&Save", CM_FILESAVE
    MenuItem  "Save&As", CM_FILESAVEAS
  END
  MenuItem  "&Help", CM_HELP
END
```

Figure 4.1 shows the appearance of this menu (resource ID "COMMANDS"), including Help (menu ID CM_HELP) and File items, the latter with the subitems New, Open, Save, and SaveAs (menu IDs CM_FILENEW, CM_FILEOPEN, CM_FILESAVE and CM_FILESAVEAS, respectively). Top-level menus that have subitems, such as File, do not have menu IDs, and selecting them causes no action besides displaying their subitems. If a top-level menu item has no subitems, as in Help, it has a menu ID and can cause some action.

Figure 4.1
The steps application with a menu resource



Because the menu is designed outside of the ObjectWindows program and because menu handling is fairly simple, you don't need to make a menu into an object. Instead, make it an attribute of the main window, much like the main window's caption. In Step 7, you will attach the menu to the main window.

Step 7: A menu for the main window

As stated previously, an application's menu is not a separate object, owned by the main window. In fact, it's not an object at all, but merely an attribute of the main window. It is stored in the *Menu* field of *Attr*, a window object data member that contains the window's *creation attributes*. You set the menu attribute, among others, in the constructor for your window by calling **AssignMenu**.

The menu resource stored in STEPS.RC has a resource ID of "COMMANDS." The call to **AssignMenu** looks like:

```
AssignMenu("COMMANDS");
```

If you're using integer resource IDs for your menu resource, you can use the overloaded **AssignMenu** member function as follows:

```
AssignMenu (100);
```

Here's how **TMyWindow**'s constructor should look. Notice that **TWindow**'s constructor is invoked to perform the initialization required of all window objects.

This code fragment is from STEP7.CPP.

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                     TWindow(AParent, ATitle)
{
  AssignMenu("COMMANDS");
  ButtonDown = FALSE;
  PenSize = 1;
  ThePen = CreatePen(PS_SOLID, PenSize, 0);
  Points = new TPointArray(50, 0, 50);
}
```

Now, when the main window is displayed, it has the menu shown in Figure 4.1. In order to make the menu selections do something, however, you must follow the remaining steps to respond to the selections from the menu.

Intercepting a menu message

When the user selects an item from a window's menu, or uses its accelerator, the window receives a Windows *command message*. To process a command message sent when a particular menu item is selected, define an ObjectWindows *command response member function*. Use the special function declaration for your command response member functions, as in the following example:

```
virtual void CMFileNew(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
```

Message ranges and offsets are explained more thoroughly in Chapter 7.

where CM_FIRST is an ObjectWindows-defined constant and a constant that represents the ID of the corresponding menu item. In this example, CM_FILENEW is defined in owlrc.h so we don't need to define it. However, if you define your own menu items, as we will for Help, you might want to define a constant such as #define CM_HELP 201. Put all these resource ID constants in a header file that you can include in your .CPP and .RC files. Note the special CM_FIRST extension used to identify a command response member function. Do not confuse this extension with the WM_FIRST extension used to identify window message response member functions.

All response member functions are passed a **TMessage** structure reference (type **RTMessage**) as a parameter. However, this structure is not often used in a command response member function. *Msg.WParam* contains the ID of the menu that was selected, which you're already aware of. You'll normally ignore

Msg.LP.Hi, which contains a 1 if the message was sent when the menu item's accelerator was used. *Msg.LP.Lo* always contains 0.

Now, you can declare all of the command message response member functions using the resource ID constants we discussed above. Note that CM_HELP is defined in steps.h, not owlrc.h.

This code fragment is from STEP7.CPP.

```
virtual void CMFileNew(RTMessage Msg) = [CM_FIRST + CM_FILENEW];
virtual void CMFileOpen(RTMessage Msg) = [CM_FIRST + CM_FILEOPEN];
virtual void CMFileSave(RTMessage Msg) = [CM_FIRST + CM_FILESAVE];
virtual void CMFileSaveAs(RTMessage Msg) = [CM_FIRST +
                                            CM_FILESAVEAS];
virtual void CMHelp(RTMessage Msg) = [CM_FIRST + CM_HELP];
```

Responding to the menu message

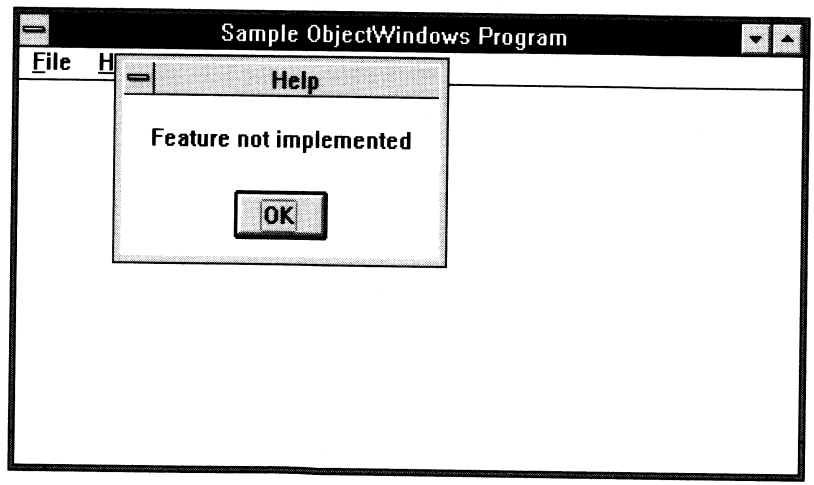
For each menu selection, you now have a member function that will be invoked. For example, for selection of a Help menu item, your **CMHelp** member function will be invoked. For now, you'll merely display a message box:

This code fragment is from STEP7.CPP.

```
void TMyWindow::CMHelp(RTMessage)
{
  MessageBox(HWindow, "Feature not implemented", "Help", MB_OK);
}
```

Figure 4.2 shows the steps application's response to the Help selection.

Figure 4.2
The steps application with help system



At this point, you can respond to the File | New menu selection by clearing the screen. Add the following **CMFileNew** member function:

This code fragment is from STEP7.CPP.

```
void TMyWindow::CMFileNew(RTMessage)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
}
```

This member function deletes all the stored points and forces a repainting of the screen. The window becomes blank because there are no points to redraw.

For the CM_OPEN, CM_SAVE, and CM_SAVEAS messages, write dummy member functions similar to **CMHelp**. Later you'll rewrite these member functions to perform meaningful actions.

The complete source code to the steps application up to this point is in the file STEP7.CPP.

C H A P T E R

5

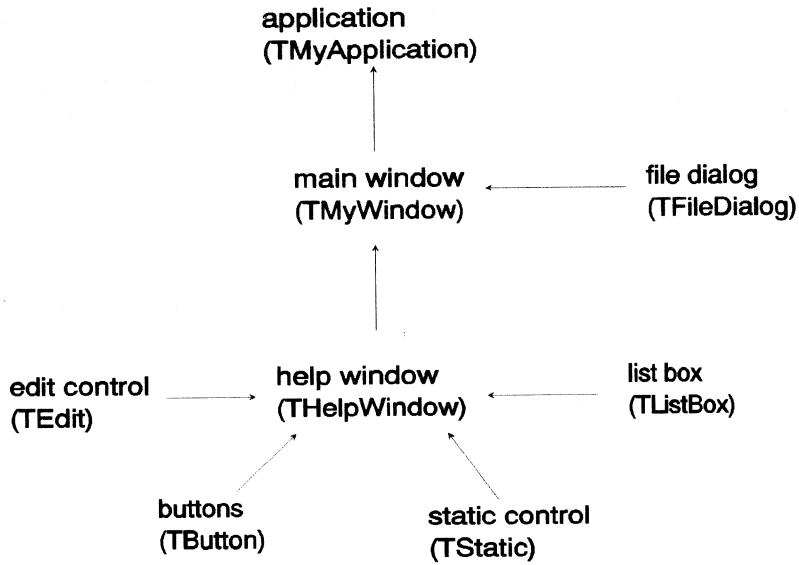
Holding a dialog

A full-featured Windows program could have many different types of window objects (such as windows, controls, and dialogs) associated with it. With the exception of the main window, each window, called a child window, has a single parent window. A single parent window can have many child windows.

These parental relationships result in a network of related parent and child windows for each application, with the main window as the ultimate parent. In this network, most windows serve as a child to one window and as a parent to others. The group of related child and parent windows for the steps application, which is now only partly implemented, is shown in Figure 5.1.

There are two types of child windows. One type is an independent child window. This type controls its own appearance and location. Message boxes and pop-up windows are independent child windows. The other type is a dependent child window. This type appears to be stuck to the surface of its parent and moves around with the parent. Controls, such as the buttons on message boxes, and some other types of child windows are dependent child windows. In this step, you'll create two types of independent child window objects: pop-up windows and dialog boxes.

Figure 5.1
A group of related parent and child windows



Parent and child are *owner* relationships. For example, in Step 8, the main window *owns* the help window. This ownership concept in Windows is almost identical to the ownership, or instance-linkage, concept of object-oriented programming. In fact, throughout this tutorial, child window objects will generally be stored in data members of their parent window objects. Although this is not required, it is a good design technique and results in clear and easily maintained programs.

Step 8: Adding a pop-up window

Let's add to the steps application a pop-up window that appears as a result of selecting the Help menu. This window can serve as the basis of a help system for the steps application. For now, you'll make the pop-up window an instance of **TWindow**. In Step 10, you'll create a new class for it and give it some useful behavior.

Since this is the first "new" window you've added (other than the main window), take a look at how window objects and window elements are created and displayed.

Creating the pop-up window

In Step 7, you responded to the Help menu selection by putting up a message box. Here, you'll substitute that message box with a blank pop-up window object, *HelpWindow*. While the immediate results will be less interesting than the message box, you'll add a lot more behavior to the help window in Step 10.

For now, just redefine **CMHelp** to create the help window:

This code fragment comes from STEP8.CPP.

```
void TMyWindow::CMHelp(RTMessage)
{
  PTWindow HelpWindow;

  HelpWindow = new TWindow(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;
  GetApplication()->MakeWindow(HelpWindow);
}
```

See "Initializing window objects" on page 120 for detailed information about creating window objects.

The MakeWindow function

The **TModule** object defines a very important member function called **MakeWindow**. **MakeWindow** performs all the actions necessary to associate an interface element with a window object safely. By "safely," we mean that **MakeWindow** checks for error conditions *before* they can cause serious problems like hanging your system. There are two important steps taken by **MakeWindow**.

1. The first is a test of the validity of the interface object, through a call to **ValidWindow**, which checks to make sure your application didn't run out of memory or otherwise fail to construct the window object completely.
2. If the interface object was constructed successfully, **MakeWindow** attempts to create and associate a window element through a call to the object's **Create** member function. **Create** uses the information in the object's data members to tell Windows what kind of interface element to create. If the element cannot be created, an error message box will pop up.

If either the validity check or the interface element creation fails, **MakeWindow** deletes the invalid window object and returns NULL. Otherwise, it returns a pointer to the interface object passed as its parameter.

Calling **MakeWindow** is the safest way to create interface elements. Calling **Create** directly without checking memory and trapping errors is dangerous, and not advisable.

Adding a dialog box

A dialog box is like a pop-up window, but it usually stays on the screen for a short period of time and performs one particular input-related task, such as choosing a printer or setting up a document page. Here, you'll add to the steps application a dialog box for opening and saving files.

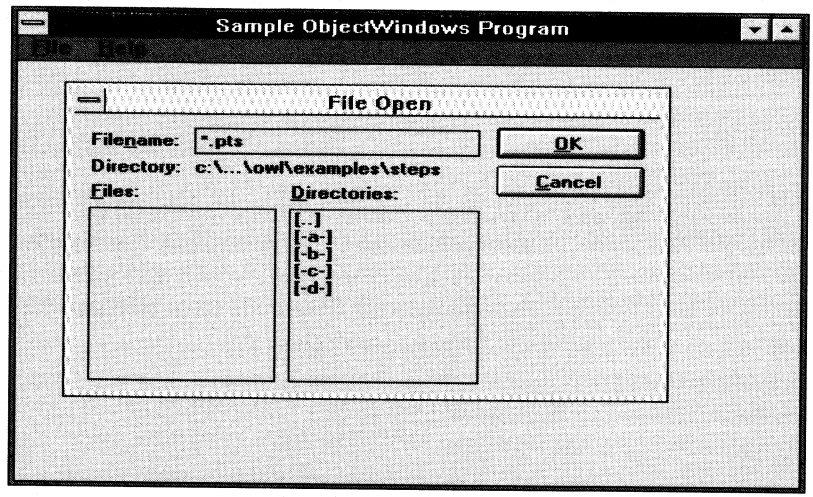
Like a pop-up window, a dialog box is an independent child window. Conceptually, adding a dialog box as a child window is exactly like adding a pop-up window, as you did earlier. A dialog box is a lot like a window, but it does have some major differences:

- Dialog classes descend from class **TDialog** rather than **TWindow**. Both **TDialog** and **TWindow** descend from **TWindowsObject**.
- Dialog boxes normally require resources that specify their size, location, and appearance.
- Dialog boxes usually perform a short task and return a value. For example, the **CanClose** message box, a limited type of dialog you created in Step 2, returned a yes or no answer from the user. Also see the line thickness dialog box you created in Step 5.

You'll add one of ObjectWindows' stock dialog boxes, the file dialog box, defined by class **TFileDialog**, a class derived from **TDialog**. The file dialog box is useful in any situation where you are asking the user to pick a disk file for saving or loading. For example, a word processing application would use a file dialog box for opening and saving documents. You will bring up a file dialog box in response to the user's selection of File | Open or File | Save As. The file dialog box will replace the "Feature not implemented" message box. In Step 9, you'll hook it up to some real files and save and open them to store and retrieve real data. For

now, you'll just show the dialog boxes. Figure 5.2 shows the appearance of the file dialog box.

Figure 5.2
The steps application with the file dialog box



Adding a data member

Instead of storing an entire file dialog box object as a data member of its parent window, construct a new file dialog box object each time you need one. What you'll store instead is the data you want to use with the file dialog box: a file name. This is good practice. Instead of keeping entire objects around when you may never need them, simply store the data you need to initialize the objects, when you need them.

Constructing a file dialog box take three parameters: a parent window, a resource identifier (ID), and a file name. The resource ID passed is the identifier of either the standard "open" or "save" file dialog (SD_FILEOPEN or SD_FILESAVE). The file name selected will be returned in the supplied file name parameter. This parameter is also used to pass a default file name for a save file dialog, and is used to pass the default file mask for an open file dialog.

The **TMyWindow** declaration now looks like this:

```
class TMyWindow : public TWindow
{
:
  char FileName[MAXPATH];
:
};
```

Running the dialog box

Depending on the resource ID passed to the **TFileDialog** constructor, the dialog box can support opening or saving files. Either option produces a dialog box similar to the one shown in Figure 5.2. There are two differences between the file open and the file save dialog:

1. The open file dialog has buttons that read OK and Cancel, while the save file dialog has Save and Cancel buttons.
2. The file save dialog initially shows the current file name in the edit area of the dialog's combo box.

Here's how you'll rewrite **CMFileOpen** and **CMFileSaveAs**:

This code fragment comes from STEP8.CPP.

```
void TMyWindow::CMFileOpen(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                    SD_FILEOPEN, strcpy(FileName,
                                    "*.PTS"))) == IDOK )
     MessageBox(HWindow, FileName, "Open the file:", MB_OK);
}

void TMyWindow::CMFileSaveAs(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                    SD_FILESAVE, FileName)) == IDOK )
      MessageBox(HWindow, FileName, "Save the file:", MB_OK);
}
```

The full source code to the steps application, to this point, is in the file STEP8.CPP.

Just as you used **TApplication::MakeWindow** to create "safe" interface elements, use **TApplication::ExecDialog** to create and execute safe dialog boxes. Like **MakeWindow**, **ExecDialog** calls **ValidWindow** to make sure that the construction of the dialog box object was successful. If it was, then **ExecDialog** calls the dialog box object's **Execute** member function. After the dialog box executes, **ExecDialog** returns the value returned by **Execute**,

unless an error occurred, in which case it returns IDCANCEL, so it looks to your program as if the user canceled the dialog box. In any case, **ExecDialog** deletes the dialog box object from memory.

Using **ExecDialog** is the safest way to run dialog boxes. You can call **Execute** directly, but doing so is dangerous, and you risk hanging your system.

Step 9: Storing the drawing in a file

Now that **TMyWindow** has a data representation of its current line (its *Points* data member), you should be able to transfer that data into a file (actually, a file stream) and read it back. In this step, you'll add data members to hold status information, and you'll modify the member functions that open and save a .PTS file.

Monitoring the status

You need to monitor two characteristics of the drawing: whether the file needs saving and whether there is a file currently loaded. You can think of these characteristics as **BOOL** attributes of **TMyWindow**, so make them data members:

```
class TMyWindow : public TWindow
{
:
  BOOL IsDirty, IsNewFile;
:
};
```

IsDirty is TRUE if the current drawing is "dirty." (Dirty means that it needs to be saved because it has changed since it was last saved or because it has never been saved.) When the user starts drawing (**WMLButtonDown**), you should set *IsDirty* to TRUE. When the user opens a new file or saves the existing one, you should set *IsDirty* to FALSE. When the user closes the application (**CanClose**), you should check the status of *IsDirty* and display the message box only if it is TRUE.

IsNewFile is TRUE only when the window is first displayed and immediately after the user selects the File | New menu (**CMFileNew**). It is set to FALSE whenever a file is opened (**CMFileOpen**) or saved (**CMFileSave** or **CMFileSaveAs**). **CMFileSave** uses **IsNewFile** to see if the file can be saved

immediately (FALSE) or if the user needs to select a file name from a file dialog (TRUE).

Listed here are the **CanClose** member function and the file saving and loading member functions. At this point, they do everything but save and load files. The file-saving behavior has been concentrated into one new member function, called **SaveFile**, and file opening is delegated to **OpenFile**.

These code fragments are from STEP9.CPP.

```
BOOL TMyWindow::CanClose()
{
  if ( !IsDirty )
    return TRUE;
  return MessageBox(HWindow, "Do you want to save?", "Drawing has
                    changed", MB_YESNO | MB_ICONQUESTION) == IDNO;
}

void TMyWindow::CMFileNew(RTMessage)
{
  Points->DeleteAll();
  InvalidateRect(HWindow, NULL, TRUE);
  IsDirty = FALSE;
  IsNewFile = TRUE;
}

void TMyWindow::CMFileOpen(RTMessage)
{
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                    SD_FILEOPEN, strcpy(FileName,
                                    "*.PTS"))) == IDOK )
      OpenFile();
}

void TMyWindow::CMFileSave(RTMessage)
{
  if ( IsNewFile )
    SaveFileAs();
  else SaveFile();
}

void TMyWindow::SaveFileAs()
{
  if ( IsNewFile )
    strcpy(FileName, "");
  if ( GetApplication()->ExecDialog(new TFileDialog(this,
                                    SD_FILESAVE, strcpy(FileName,
                                    "*.PTS"))) == IDOK )
      SaveFile();
}
```

```
void TMyWindow::CMFileSaveAs(RTMessage)
{
  SaveFileAs();
}

void TMyWindow::SaveFile()
{
   ofpstream os(FileName);

   os << Points;
   os.close();
   IsNewFile = IsDirty = FALSE;
}

void TMyWindow::OpenFile()
{
   ifpstream is(FileName);
   if ( is.bad() )
     MessageBox(HWindow, "Unable to open file", "File Error", MB_OK |
                MB_ICONEXCLAMATION);
   else
   {
     Points->DeleteAll();
     is >> Points;
     is.close();
     IsNewFile = IsDirty = FALSE;
     InvalidateRect(HWindow, NULL, 1);
   }
}
```

Opening and saving files

You've built the framework for opening and saving .PTS files. But the real work, the actual reading and writing of the *Points* array, remains. You can get the job done quickly by making *Points* a *streamable* array.

The first step is to multiply inherit from both **Array** and **TStreamable**.

This code fragment is from STEP9.CPP.

```
_CLASSDEF(TPointArray)
class TPointArray : public Array, public TStreamable
{
public:
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0)
    : Array(upper, lower, aDelta){};
⋮
protected:
  virtual void write( Ropstream );
  virtual Pvoid read( Ripstream );
⋮
};
```

TPointArray's **write** and **read** member functions are invoked when a request is made to read or write a **TPointArray**. These member functions, listed next, perform the actual reading and writing of the points in a **TPointArray**:

These code fragments are from STEP9.CPP.

```
Pvoid TMyWindow::read(Ripstream is)
{
  TWindow::read(is);
  is >> Points;
  return this;
}

void TMyWindow::write(Ropstream os)
{
  TWindow::write(os);
  os << Points;
}
```

Various other member functions must be defined before **TPointArray** is truly streamable. The remaining methods, a handful of which are rote "one-liners," can be found in the STEP9.CPP program in the EXAMPLES\STEPS subdirectory. You'll be looking closely at all of these streamable member functions in Chapter 15. Just skim the implementation of these methods, for now. Do look at **TMyWindow**'s **read** and **write** member functions to see how making **TMyWindow** streamable is a simple matter once **TPointArray** is streamable.

C H A P T E R

6

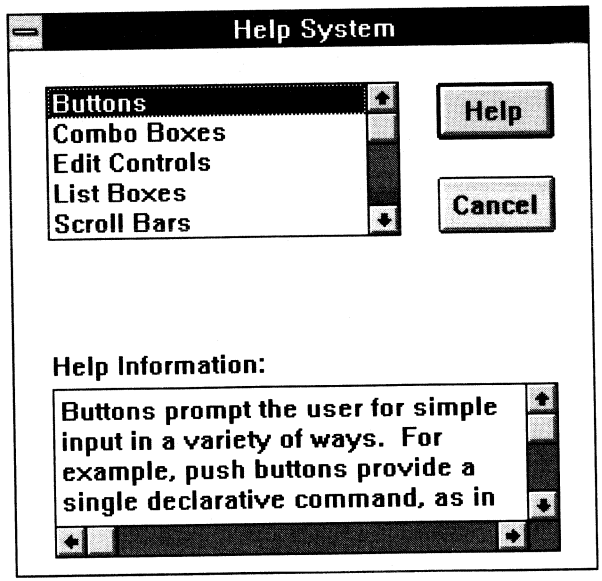
Popping up windows

Now you're ready to add the finishing touch to your Object-Windows program: The help system. You can think of a help system as a program within a program that can be invoked at a moment's notice. One of the benefits of a Windows program is that, with its multiple windows, it can encompass and unify many activities. Adding a help system to the steps application will illustrate how to design ObjectWindows programs in a modular format for a high degree of reusability and maintainability.

Step 10: Popping up a help window

In Step 8, you added a Help menu to the steps application's main window. Selecting this menu produced a plain window, an instance of **TWindow**. In this step, you'll replace that plain window with a useful **THelpWindow** that makes use of some of the Windows controls supported by ObjectWindows: list boxes, scroll bars, edit controls, static controls, buttons, and combo boxes. In the process, you will define the help system in a separate module, requiring little change in your existing program. Figure 6.1 shows the finished help system, which provides a small description of some of the controls ObjectWindows supports.

Figure 6.1
The steps application's help system



Using modules with ObjectWindows

A module is a convenient storage mechanism for definitions of a class's member functions. This is especially important in Windows programming, where you tend to reuse windows (window objects in ObjectWindows) from one application to the next. Here, you'll build your help system as a class called **THelpWindow** and store its definition in a file called HELPWIND.CPP.

Modifying the main program

The steps application requires only two changes:

- Add the HELPWIND.CPP module to the the steps application project or make file.
- Remove all of the code to set up the attributes of HELPWIND.CPP from **TMyWindow::CMHelp**. This code will now go into **THelpWindow**'s constructor.

Now, when the user selects the Help menu item, the steps application responds by producing a **THelpWindow**. That means you need to define the **THelpWindow** class. You'll do that in the HELPWIND.CPP module.

Creating the module

The **HelpWind** module will do nothing more than define the interface and implementation of the class **THelpWindow**. Once created, this module can be used by any other ObjectWindows program. It is, in effect, a window in a module. Here is the format of the **HelpWind** module (the gaps will be filled in later):

This code is in helpwind.h.

```
#ifndef __HELPWIND_H
#define __HELPWIND_H

#ifndef __OWL_H
#include <owl.h>
#endif

#ifndef __LISTBOX_H
#include <listbox.h>
#endif

#ifndef __EDIT_H
#include <edit.h>
#endif

#define ID_LISTBOX  101
#define ID_BUTTON1  102
#define ID_BUTTON2  103
#define ID_EDIT     104

_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
  THelpWindow(PTWindowsObject AParent);
  virtual void SetupWindow();
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
  virtual void FillEdit(Pchar SelString);
};

#endif
```

This code is in HELPWIND.CPP.

```
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
```

```
#include <string.h>
#include <owl.h>
#include <static.h>
#include <edit.h>
#include <listbox.h>
#include <button.h>
#include "helpwind.h"

THelpWindow::THelpWindow(PTWindowsObject AParent) :
   TWindow(AParent, "Help System")
{
:
}

void THelpWindow::SetupWindow()
{
:
}

void THelpWindow::HandleListBoxMsg(RTMessage Msg)
{
:
}

void THelpWindow::HandleButton1Msg(RTMessage)
{
:
}

void THelpWindow::HandleButton2Msg(RTMessage)
{
:
}

void THelpWindow::FillEdit(Pchar SelString)
{
:
}
```

Adding controls to a window

In Step 8, you learned there are two types of child windows: dependent and independent. You also added to the main window two types of independent child windows: dialogs and pop-up windows. Now, you'll add *controls*, which are a type of dependent child window. These controls will have the help window as a parent window. Remember, the help window is an independent child window whose parent window is your application's main

window. Therefore, your help window is simultaneously a child window and a parent window.

What are controls?

Controls are visual devices that make up part or all of a window's or dialog box's user interface. For example, the OK and Cancel buttons on file dialog boxes are controls, as are the list boxes used to display files. The steps application's main window has no controls, but here you'll add some to your help window.

There are many types of controls, including list boxes, scroll bars, edit controls, static controls, buttons, and combo boxes. Your help window, pictured in Figure 6.1, makes use of a list box (the large box in the top half of the window), an edit control (the large box in the lower half of the window), a static control (the text "Help Information:"), and two buttons (Help and Cancel). Scroll bars are part of both the edit control and the list box, but these are not scroll bar objects. Scroll bars are most often used as optional parts of other controls and windows and rarely appear as standalone controls.

The ObjectWindows **TControl** class supplies behavior shared by all control window objects. Its derived classes define behavior unique to the type of control windows that their instances represent. For example, **TEdit** and **TListBox** define behavior unique to edit controls and list boxes, respectively. You should also note that **TControl** descends from **TWindow**. Controls are actually specialized windows.

Creating window controls

You can associate a resource-defined control with an ObjectWindows object. See "Associating control objects" in Chapter 11.

While they behave identically, there's an important programming difference between the controls in dialog boxes, such as file dialogs, and the controls in windows, such as in your help window. You specify the characteristics of a dialog's controls in the dialog's resource definition. Windows creates the controls of a dialog using its resource definition; the controls are not necessarily associated with ObjectWindows objects. When a dialog's controls are not associated with ObjectWindows objects, the dialog must manage them through direct calls to Windows functions.

A window's controls, however, are always represented by ObjectWindows objects. A parent window manages its controls through

the rich set of member functions defined by the ObjectWindows control objects. For example, to get the text item the user has selected from a list box, call the list box object's **GetSelString** member function. Like a window or dialog box object, a control object has a corresponding visual element. A control object and its control element are linked through the control object's *HWindow* data member, as are windows and dialogs. Each control has a unique ID that is used by its parent window to identify the control for routing of control events, such as when the user clicks a button. For clarity, you should define constants for each control ID:

```
#define ID_LISTBOX 101      // ID for list box 1
#define ID_BUTTON1 102      // ID for first (OK) button
#define ID_BUTTON2 103      // ID for second (Cancel) button
#define ID_EDIT 104         // ID for edit control
```

Control objects as data members

It is sometimes convenient to store a pointer to a control object (or other child window) as a data member in a window object. This is only necessary for child windows that will later be manipulated directly by calling their object member functions. In this case, only the list box and edit control qualify. **THelpWindow** will store each of these control objects in a separate data member. Here is part of **THelpWindow**'s object definition:

This code fragment is from helpwind.h.

```
_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
:
};
```

Once these child control objects have been instantiated, you can manipulate them with member function calls. For example, you can add a string to the list in *ListBox* by calling **ListBox-> AddString**. It is possible to access the control objects for child windows not stored as data members by using the **TWindowsObject** member functions **Next** and **Previous**, but it is far more convenient to do so with data members.

Managing controls

Any window class that has control objects must define a constructor to construct its control objects. It can also redefine **SetupWindow**, which it inherits from **TWindow**, to set up its control objects. **THelpWindow** redefines **SetupWindow** to set up the list, and set the selection of its child listbox. Note that **THelpWindow::SetupWindow** first calls **TWindow::SetupWindow**, which creates the windows that have been constructed as child windows of **THelpWindow**.

The following code lists the help window's constructor. The first call the constructor makes is to **DisableAutoCreate**, a call that you'll probably want to make in the constructors of your own pop-up windows. This call sets a flag that indicates that the creation of the pop-up window is not tied to the creation of its parent window. Such behavior is not normally appropriate for a pop-up.

Afterward, the **THelpWindow** constructor sets its location and size attributes. Because the **THelpWindow** is not fully functional without its child windows, the construction of a **THelpWindow** is not complete until its child windows have been constructed. The **THelpWindow** passes **this** as the first parameter to the constructor of each of its controls. (**this** is a pointer to the **THelpWindow** object being constructed.) A control ID is the second parameter in every control's constructor call. Additional parameters include location and size data. Note that *Listbox* and *Edit* are stored for later reference.

This code fragment is from HELPWIND.CPP.

```
THelpWindow::THelpWindow(PTWindowsObject AParent) :
   TWindow(AParent, "Help System")
{
  DisableAutoCreate();
  Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 300;
  Attr.H = 300;
  ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 180, 80);
  new TButton(this, ID_BUTTON1, "Help", 220, 20, 60, 30, TRUE);
  new TButton(this, ID_BUTTON2, "Cancel", 220, 70, 60, 30, FALSE);
  Edit = new TEdit(this, ID_EDIT, "", 20, 180, 260, 90, 40, TRUE);
```

```
    new TStatic(this, -1, "Help Information:", 20, 160, 160, 20, 0);
}
```

After the help window is created, **THelpWindow::SetupWindow** is called to perform setup for the help window. The help window is set up by creating its controls and initializing its list box.

THelpWindow::SetupWindow first calls **TWindow::SetupWindow**, which creates the child windows of the help window. It then calls member functions of its *ListBox* to initialize the list and selection of the list. (The controls are automatically displayed when created because WS_VISIBLE is set as a creation style by default.)

This code fragment is from HELPWIND.CPP.

```
void THelpWindow::SetupWindow()
{
  TWindow::SetupWindow();
  ListBox->AddString("List Boxes");
  ListBox->AddString("Buttons");
  ListBox->AddString("Check Boxes");
  ListBox->AddString("Radio Buttons");
  ListBox->AddString("Group Boxes");
  ListBox->AddString("Scroll Bars");
  ListBox->AddString("Edit Controls");
  ListBox->AddString("Static Controls");
  ListBox->AddString("Combo Boxes");
  ListBox->SetSelIndex(0);
};
```

Calls to **THelpWindow**'s constructor and the **SetupWindow** member function are enough to properly display the controls in the help window. The list box will scroll and buttons will press, but with no resulting action. In the next section you'll define a response to control events.

Responding to control events

You respond to menu events by defining command message response member functions, as you've already read. You respond to control events by defining child-ID-based message response member functions. To identify a child-ID-based message response member function, use the sum of the ID_FIRST constant and the ID of the control, as shown here:

This code fragment is from helpwind.h.

```
_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
```

```
  :
    virtual void HandleListBoxMsg(RTMessage Msg) =
      [ID_FIRST + ID_LISTBOX];
  :
  };
```

ID_FIRST is a constant defined by ObjectWindows. In this example, ID_LISTBOX is the identifier of a child list box control of **TMyWindow**, the parent. **HandleListBoxMsg** defines the response to all messages sent to **TMyWindow** from this list box control.

A control sends a message to its parent whenever an event occurs that its parent should be aware of. The events that result in control notification messages to the parent depend on the type of child control. A notification code is passed to the parent identifying the type of event that occurred. For example, list boxes send LBN_SELCHANGE notification messages when their selection changes, and buttons send BN_CLICKED messages when clicked.

Control response member functions are passed a *Msg* parameter, of type **RTMessage**, like command message response member functions. *Msg.LP.Hi* contains a code for the type of event that occurred. *Msg.LP.Lo* contains the window handle of the control. *Msg.WParam* contains the control's ID.

F1

Help

Lists of the codes sent with *control notification messages* appear in online Help. For clarity, give these message response member functions names like **HandleListBoxMsg**. Here's the **THelpWindow** class definition showing the member function prototypes:

This code fragment is from helpwind.h.

```
_CLASSDEF(THelpWindow)
class THelpWindow : public TWindow
{
public:
  PTListBox ListBox;
  PTEdit Edit;
  THelpWindow(PTWindowsObject AParent);
  virtual void SetupWindow();
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
  virtual void FillEdit(Pchar SelString);
};
```

Next, decide how **THelpWindow** should respond to messages from its controls. When the user clicks the Cancel button (ID_BUTTON2), the window should close. The implementation of **HandleButton2Msg** is quite simple:

```
void THelpWindow::HandleButton2Msg(RTMessage Msg) {
  CloseWindow();
}
```

Buttons send only two types of control notification messages, BN_CLICKED and BN_DBLCLICKED. It is not necessary here to check the notification code of the control message received because the window should close in either case.

When the user clicks the OK button (ID_BUTTON1), you want the steps application to check to see which list box item is selected and fill the edit control (ID_EDIT) with corresponding text. Thus, **HandleButton1Msg** calls **GetSelString** for the list box (ID_LISTBOX) to get the text of the selected list item. The text is passed to a member function you must write called **FillEdit**. **FillEdit**'s definition is available in HELPWIND.CPP.

```
void THelpWindow::HandleButton1Msg(RTMessage Msg) {
  char SelString[25];

  ListBox->GetSelString(SelString, sizeof (SelString));
  FillEdit(SelString);
}
```

As an added touch, you can respond to the message generated when the user double-clicks directly on a list box item by also filling the edit control with text. Since a list box sends a variety of messages to its parent, check the code of the message prior to filling the edit control. If the value is LBN_DBLCLK, the user double-clicked, otherwise a call to **DefWndProc** tales care of all the other messages that might come in to a list box.

See page 117 for more information about DefWndProc.

```
void THelpWindow::HandleListBoxMsg(RTMessage Msg) {
  char SelString[25];

  if (Msg.LP.Hi == LBN_DBLCLK) {
    ListBox->GetSelString(SelString, sizeof (SelString));
    FillEdit(SelString);
  }
  else DefWndProc(Msg);
}
```

PART 2

Using ObjectWindows

C H A P T E R

7

Overview

ObjectWindows is a comprehensive set of classes that streamlines your development of Microsoft Windows programs with C++.

This chapter gives an overview of the ObjectWindows class hierarchy. The remaining chapters in this part provide detailed descriptions of the different parts of the hierarchy.

In addition to describing the object hierarchy, this chapter outlines the basic principles of programming for the Windows environment, including using resources, calling the Windows API, and receiving and handling messages from Windows.

ObjectWindows conventions

The many classes and structures in the ObjectWindows hierarchy have names that start with *T* (such as **TWindow**). For all the *T* classes, ObjectWindows also defines the following related types:

- **P*class***, a pointer type to class *class*. (For example,**PTWindow** is defined as a pointer to **TWindow**.)
- **R*class***, a reference type to class *class*. (For example,**RTWindow** is defined as a reference to **TWindow**.)
- **RP*class***, a reference to a pointer to class *class*. (For example, **RPTWindow** is defined as a reference to a pointer to **TWindow**.)
- **PC*class***, a pointer type to**const** class *class*. (For example, **PCTWindow** is defined as a pointer to a **const TWindow**.)

- **RC*class***, a reference type to**const** class *class*. (For example, **RCTWindow** is defined as a reference to **const TWindow**.)

We provide these types to simplify some complicated Windows programming procedures. For example, Windows' dynamic link libraries (DLLs) would otherwise require declarations like

This is equivalent to PTWindow

```
TWindow _FAR *
```

where a pointer to a **TWindow** was required or

This is equivalent to RPTWindow

```
TWindow _FAR * _FAR &
```

for a reference to a pointer to a **TWindow**.

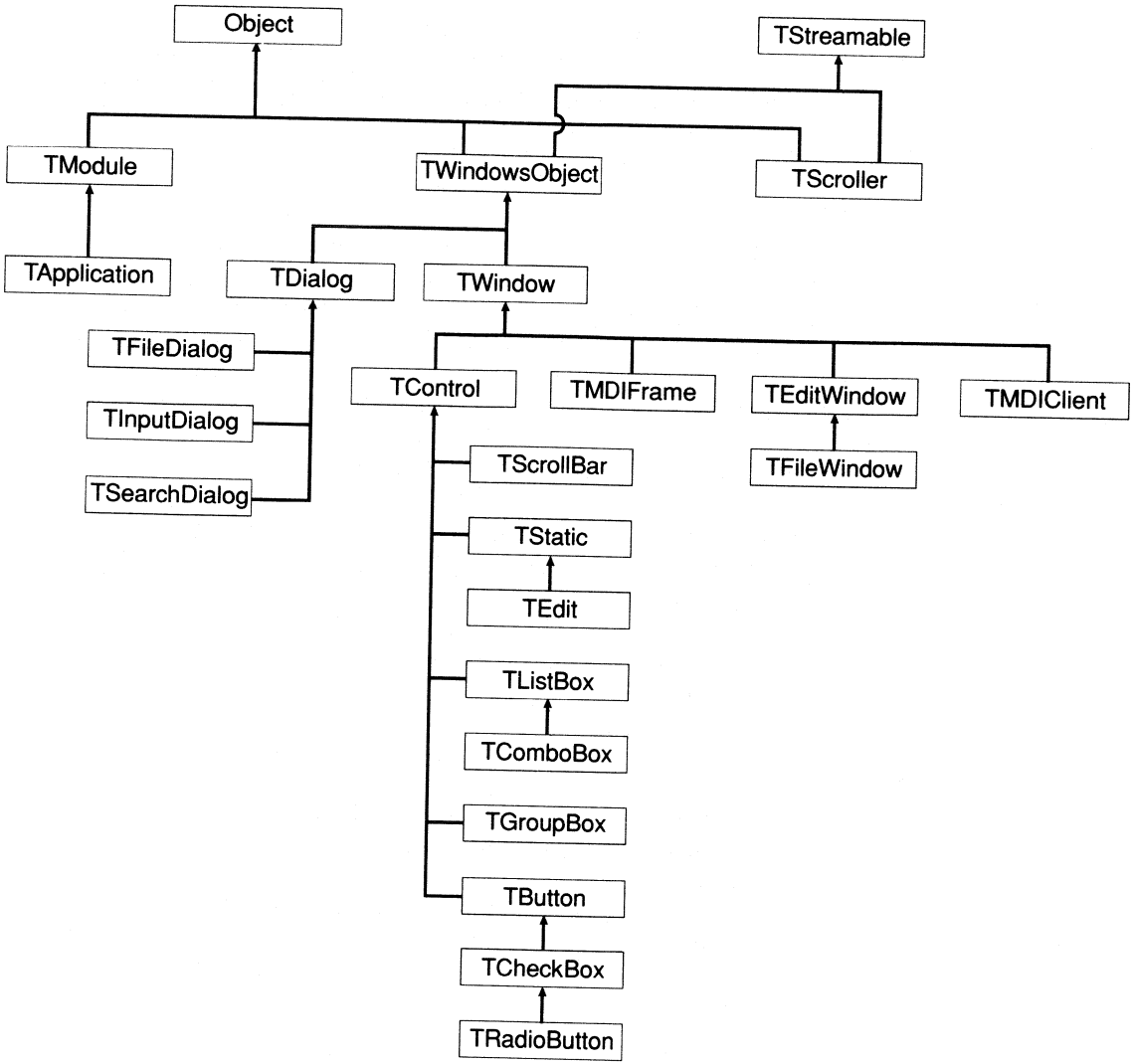
Using the *P, R, RP, PC* and *RC* types will ease coverting your application should you later decide to make portions a DLL. We use these types extensively through the ObjectWindows example programs and the steps application.

Message-response member functions are named, by convention, after the messages they respond to, but without the underscores. For example, a member function responding to the message WM_KEYDOWN would be called **WMKeyDown**, and a message responding to the CM_FILEOPEN command would be called **CMFileOpen**.

The ObjectWindows hierarchy

ObjectWindows is a library consisting of a hierarchy of classes that you can use, modify, or add to, using inheritance. Chapter 16 is a complete reference to the classes, their data members, and member functions.

Figure 7.1: ObjectWindows class hierarchy



Object

Object is the base class for all ObjectWindows derived classes and is defined in the container class library. **Object** name isn't **TObject** since it's part of the container class library, whose names don't begin with *T*.

TModule

ObjectWindows dynamic-link libraries (DLLs) construct an instance of **TModule**, which acts as an object-oriented stand-in for the library (DLL) module. ObjectWindows applications construct an instance of **TApplication**, derived from **TModule**. **TModule** defines behavior shared by both library and application modules. **TModule** member functions provide support for window memory management and error-processing.

TApplication

This class defines the behavior required of all ObjectWindows applications. Each ObjectWindows application you write will derive its own application class from **TApplication**. It is responsible for, among other things, initializing the main window object. Application objects are described in detail in Chapter 8, "Module and application objects."

Interface objects

The remaining objects in the ObjectWindows hierarchy are classified generally as interface objects. They are interface objects both in the sense that they represent elements in the Windows user interface, and because they serve as a kind of interface between your application code and the Windows environment. Interface objects are described in detail in Chapter 9, "Interface objects."

TWindowsObject

TWindowsObject is a base class that unifies the three main types of ObjectWindows interface objects: windows, dialog boxes, and controls. It provides member functions to handle the creation, message processing, and destruction of window objects.

Window objects

Window objects represent not only the familiar windows of the windowing environment, but also most of the visual elements within that environment, such as controls.

TWindow

TWindow is a general-purpose window class whose instances can represent main, pop-up, or child windows of an application. Instances of **TWindow** can display graphics, but most often you will specialize **TWindow**'s behavior in classes that you derive from **TWindow**.

TEditWindow

Derived from **TWindow**, **TEditWindow** defines a class that allows text editing in a window.

TFileWindow

Derived from **TEditWindow**, **TFileWindow** defines a class that allows text editing in a window, but can also load and save text files.

Dialog objects

Dialog objects serve to facilitate interactive groups, particularly groups of controls such as buttons, list boxes, and scroll bars. Dialog objects are explained in detail in Chapter 11, "Dialog objects."

TDialog

This class serves as a base for derived classes that manage Windows dialog boxes. Dialog objects are associated with dialog resources, and can be run as either modal or modeless dialog boxes. Member functions are provided to handle communication between a dialog and its controls.

See page 143 for the distinction between modal and modeless dialog boxes.

TFileDialog

TFileDialog is a dialog class that is immediately useful in many applications. It defines a dialog that allows the user to choose a file for any purpose, such as opening, editing, or saving.

TInputDialog

This class defines a dialog box for user input of a single text item.

Control objects

Within dialogs and some windows, controls allow users to enter data and select options. Control objects provide a consistent and simple means of dealing with all the different kinds of controls defined by Windows. Control objects are described in detail in Chapter 12, "Control objects."

TControl

See Figure 12.1 for examples of some of these controls.

TControl is an abstract class that serves as a common base class for all control objects, including list boxes and buttons. It defines member functions that create controls and process messages for its derived classes.

TButton

Instances of **TButton** represent Windows push buttons.

TCheckBox

Derived from **TButton**, **TCheckBox** instances represent Windows check boxes and provide member functions to manage their state.

TRadioButton

Derived from **TCheckBox**, **TRadioButton** instances handle creation and state management for Windows radio buttons.

TListBox

TListBox instances represent a Windows list box. This class handles creation of and selection from Windows list boxes, and defines member functions to manipulate items in a list.

TComboBox

Derived from **TListBox**, **TComboBox** defines behavior for Windows combo boxes. A combo box is an interdependent list box and edit control.

TGroupBox

Instances of **TGroupBox** represent Windows group boxes.

TStatic

TStatic provides member functions that set, query, and clear the text of a static (output only) control.

TEdit

Derived from **TStatic**, **TEdit** provides the extensive text processing capabilities for a Windows edit control.

TScrollBar **TScrollBar** defines member functions that manage the range and thumb position of a standalone scroll bar control.

MDI objects

Windows implements a standard for handling multiple windows within the framework of a single window. This standard is called the Multiple Document Interface (MDI). ObjectWindows provides a means of setting up and manipulating MDI windows. MDI objects are described in detail in Chapter 14, "MDI objects."

TMDIFrame **TMDIFrame** provides the windowing behavior appropriate for the main window of an application that follows the Windows MDI specification.

TMDIClient **TMDIClient** provides additional support for MDI windows. The MDI client object is the object that actually manages the MDI window's client area, which is the area that contains MDI windows ("documents").

Scroller objects

TScroller is the ObjectWindows object that gives life to window scroll bars, providing an automated way to scroll the text and graphics you put into your windows. **TScroller** also scrolls its owner when the user drags the mouse from the inside to the outside of a window's client area; therefore **TScroller** works for windows that don't even have scroll bars.

Windows API functions

There are 600 or so functions in the Windows API. A Windows application manipulates its environment, modifies its appearance, or acts in response to user input by calling Windows functions. Using ObjectWindows, however, you can create windows, display dialog boxes, and manipulate controls without calling most Windows functions directly.

ObjectWindows calls Windows functions

Many member functions of ObjectWindows call Windows functions. But ObjectWindows isn't duplicating functionality; it's repackaging some of Windows' functions into an object-oriented application framework. In addition, ObjectWindows greatly simplifies the task of specifying the numerous parameters required by Windows functions. Often ObjectWindows automatically supplies parameters, such as window handles and child window IDs, which are stored as data members in interface objects.

For example, many Windows functions require a handle to a window to specify which window they are to act upon, and these functions are usually called from the member functions of a window object. The object holds the handle of its associated window in its *HWindow* data member, so it can pass that as the handle, freeing you from having to specify that item each time. The window handle (and many other bits of information Windows requires) are *encapsulated* within the object. Thus, ObjectWindows's classes serve as an object-oriented interface to the non-object-oriented Windows API.

Access to Windows functions

All Windows API functions are still available to an ObjectWindows program. For example, this code calls the Windows function **MessageBox** to produce a message box:

```
Reply = MessageBox(HWindow, "Do you want to save?",
                   "File has changed", MB_YESNO | MB_ICONQUESTION);
```

Many Windows functions have important return values. In the case of **MessageBox**, the return value is an integer that the example stores in *Reply*. The value of this integer indicates the action the user took to close the message box. If the user clicked the Yes button, the result is equal to the Windows-defined constant IDYES. If the user clicked the No button, the result is IDNO.

Windows functions require you to pass, as arguments, a variety of **WORD** and **DWORD** constants. These constants represent various styles, return values, identifiers, and more. Use of these constants results in code that is more readable, maintainable, and independent of changes in future versions of Windows. For example, it is

more informative to define a message box with the type MB_YESNO | MB_ICONQUESTION rather than 0x0024. MB_YESNO, IDYES, and the other constants are defined in windows.h.

If you look in windows.h you will see that the function **MessageBox** is declared as

```
int  FAR PASCAL MessageBox(HWND, LPSTR, LPSTR, WORD);
```

You don't need to include windows.h in your application; it's automatically included by the file owl.h.

The word *int* should be familiar to you. But what do **FAR**, **PASCAL**, **HWND**, **LPSTR**, and **WORD** mean? They are all new types defined in windows.h. **FAR** and **PASCAL** are the same as **far** and **pascal** (windows.h also defines uppercase equivalents for the data types **near**, **void**, and **long**). All Windows API functions are type **FAR PASCAL** functions with the exception of **wsprintf**, which is type **FAR cdecl**.

F1 *Help*

Some functions require more complex data structures, such as those describing fonts (**LOGFONT**) or window classes (**WNDCLASS**). Usually you pass pointers to these structures as parameters in Windows functions that require them. For a list of available structures see the online Help.

Combining style constants

F1 *Help*

Windows functions that produce interface elements usually require some style parameter of type **WORD** or **DWORD**. Windows defines hundreds of style constants, which are listed in the online Help. Style-constant identifiers consist of a two-letter mnemonic prefix followed by an underscore and a descriptive name. For example, WS_POPUP is a window style constant ("WS_" means "window style") for pop-up windows.

Often, these styles are combined using the bitwise OR operator | to produce another style. In the **MessageBox** example, you pass MB_YESNO | MB_ICONQUESTION as the style parameter. This style produces a message box with two buttons, Yes and No, and a question mark icon.

Keep in mind that some styles are meant to be mutually exclusive. Combining these styles produces unpredictable, unintended, and probably undesirable results.

Types of Windows functions

This section discusses the kinds of Windows functions available to your ObjectWindows programs.

Window manager interface functions

These functions handle messages, manipulate windows and dialog boxes, and create system output. This category includes functions for menus, cursors, and the Clipboard.

Graphics device interface (GDI) functions

These functions output text, graphics, and bitmaps on a variety of devices, including the screen and the printer. The functions are not tied to any particular device, but are device independent.

System services interface functions

These functions handle a wide range of system services, including memory management, interfacing with the operating system, resource management, and communications.

Callback functions

Callback functions are user-supplied functions that are called by routines external to your program. Typically, Windows will invoke one of your callback functions as a result of a request to enumerate objects maintained within Windows, such as fonts or windows. When you make such a request of Windows, you must supply the address of the callback to Windows so that Windows can "call you back" by invoking your callback function.

When a callback function is invoked, as mentioned earlier, it is called in a context that is different from your program—usually from inside Windows. Thus, in order for the callback function to have access to the global variables defined in your program (and residing in your program's data segment), a means must be provided to ensure the data segment is set correctly to the data segment that corresponds to your program. The means used within Windows to accomplish this are called "instance thunks." Instead of passing the actual address of your callback function to Windows, you pass the address of an instance thunk. An instance thunk merely sets up the data segment register to point to your data segment and then transfers control to your callback function. Instance thunks are created for callback functions through the Windows function **MakeProcInstance**, as shown here:

```
FARPROC ThunkPtr = MakeProcInstance((FARPROC) MyCallback,
                                    GetApplication()->hInstance);
```

The first argument to **MakeProcInstance** is a pointer to your callback function. The second is the instance handle of your application.

All callback functions must be exported using the **_export** keyword. For example, you might declare a callback function as

```
BOOL FAR PASCAL _export MyCallBack(HWND hWnd, DWORD lParam);
```

All Windows callback functions assume the Pascal calling sequence, and are thus declared with the **PASCAL** keyword.

Because Windows callback functions are called from outside the ObjectWindows environment and are not being invoked from a C++-aware environment, it is not possible to use non-static C++ member functions as callback functions. Non-static member functions make use of a hidden parameter (**this**) that Windows will not supply. Static member functions, however, do not use the hidden **this** pointer and thus are usable as callback functions. Be sure to use the **PASCAL** keyword when declaring these static member callback functions if Windows will be invoking them.

Enumeration functions

The most frequently used Windows callback functions will be those invoked as a result of Windows enumeration functions. These functions permit you to *enumerate*, or loop over, certain types of elements in the Windows system. For example, you might want to enumerate over the fonts in the system and print a sample of text in each. To use these functions, you must supply the address of your callback function to Windows in the manner described previously. Windows functions that require callback functions include **EnumChildWindows**, **EnumClipboardFormats**, **EnumFonts**, **EnumMetaFile**, **EnumObjects**, **EnumProps**, **EnumTaskWindows**, and **EnumWindows**.

As an example, let's say you have a C++ static member function called **TMyWindow::ActOnWindow** that is declared as follows:

```
class TMyWindow
{
:
  static BOOL FAR PASCAL _export TMyWindow::ActOnWindow(HWND hWnd,
                                                        DWORD lParam);
:
};
```

You could then pass this function as a callback in a call to the Windows function **EnumWindows**:

```
FARPROC lpEnumWinFunc;
⋮
lpEnumWinFunc = MakeProcInstance((FARPROC) TMyWindow::ActOnWindow,
                                 GetApplication()->hInstance);

ReturnVal = EnumWindow(lpEnumWinFunc, aLongVal);
FreeProcInstance(lpEnumWinFunc);
```

Callbacks for enumeration functions must return nonzero to continue enumeration or zero to stop it. In the previous example, the static member function **TMyWindow::ActOnWindow** would take some action on the window specified by the supplied handle. The *aLongVal* parameter is any value the caller of **EnumWindows** chooses to pass to the callback function.

Using smart callbacks

The extra steps of calling **MakeProcInstance** and **FreeProcInstance** can be avoided if you compile your code using the "smart callbacks" compile option. This option modifies the prolog and epilog code to assume that DS == SS; in other words, that the default data segment is the same as the stack segment; this eliminates the need for the **MakeProcInstance** and **FreeProcInstance** calls. The previous example, if compiled with smart callbacks, would become

```
ReturnVal = EnumWindow((FARPROC) TMyWindow::ActOnWindow, aLongVal);
```

Windows messages

When Windows determines that an event has occurred that affects an application, it sends the application a message. There are many different types of messages coming from a variety of sources:

- The user, by moving or clicking the mouse or typing on the keyboard, generates a user-event message.
- Your program can send messages directly to itself.
- Your program can call Windows functions that result in its receiving other Windows messages.

- Applications can send messages to other applications through dynamic data exchange (DDE).
- Windows itself can send a variety of messages to your application.

A Windows application is responsible for retrieving and responding to the messages that are sent to it. In the process, the application might invoke a function that has been defined to respond when a particular event occurs that affects a particular window.

ObjectWindows handles this retrieval and routing of messages for you. ObjectWindows also supplies default processing for incoming messages. You need only define responses that are unique to your application. In doing so, you declare and define special *message response member functions* in the classes you derive from the supplied ObjectWindows classes.

The following is a sample declaration for a member function that responds to a click the left button of the mouse, as it appears in the definition of a **TMyWindow** class derived from the ObjectWindows **TWindow** class.

```
class TMyWindow : public TWindow {
public:
⋮
  virtual void WMLButtonDown(RTMessage Msg) =
    [WM_FIRST + WB_LBUTTONDOWN];
⋮
};
```

The form of the declaration shown previously is shared by all message response member functions. C++ method member functions declarations have been extended to allow an integer identifier to be associated with a virtual method member functions. With this aid, ObjectWindows is able to map your application's messages to your response member functions.

For example, if you want an instance of **TMyWindow** to beep when it's clicked, simply define this behavior in a message response member function:

```
void TMyWindow::WMLButtonDown(RTMessage Msg)
{
  MessageBeep(0);       // a Windows function call
}
```

It's that easy.

Windows message parameters

Four parameters are passed when a Windows application receives a message that has been dispatched to it. Two of these are the handle (the Windows identifier) of the affected window and the message identifier. Your application has no need for these two parameters because ObjectWindows handles the mapping of an incoming message to a window object's response member function. The other two parameters, **WORD** and **LONG** values, hold useful data. These parameters are packaged as the *WParam* and *LParam* members in an ObjectWindows-defined **TMessage** structure, a reference to which (type **RTMessage**) is passed to your message response member functions. Note that the **TMessage** structure also contains a *Result* member, set internally by ObjectWindows, which you'll normally ignore.

WParam is a **WORD** that usually holds one piece of information, often a handle, identifier, or code. For example, a menu selection generates a WM_COMMAND message whose *WParam* equals the ID of the menu item selected.

LParam may contain a **LONG** value, such as a far pointer. Or *LParam* may hold two pieces of information, one in its high-order word and one in its low-order word. As an example, the *LParam* passed to a WM_LBUTTONDOWN ("left-button-down") response member function contains the x- and y-coordinates of the position at which the mouse was clicked.

For convenient access to the high- and low-order **WORD**s, *LParam* has been declared as a member of a union along with an *LP* structure containing *Lo* and *Hi* **WORD** members. Assuming you've declared *Msg* as a **TMessage** structure, you can therefore use *Msg.LP.Hi* and *Msg.LP.Lo*. to refer to the high- and low-order words of *Msg.LParam*. *TMessage::WParam* is similarly declared as a member of a union along with a *WP* structure, with *Hi* and *Lo* members that are **BYTE**s, not **WORD**s.

Types of Windows messages

The following section lists the types of messages a Windows application can receive. For a complete list, see the online Help.

Window management messages

These messages signal the state of a window has changed. For example, WM_CLOSE is sent when a window is closed and WM_PAINT is sent when part or all of a window needs to be redisplayed.

Initialization messages

These messages, including WM_CREATE and WM_INITDIALOG, are sent when a program creates a window or a dialog box.

Input messages

These messages are received as a result of input through the mouse, keyboard, scroll bars, or system timer. One of the most common input messages is WM_COMMAND, which results from a menu or accelerator selection, or from a control event, such as clicking a button.

System messages

These messages are sent in response to the user's manipulation of an application's standard accessories, such as its Control-menu box, scroll bars, or minimize or maximize button. The message WM_SYSCOMMAND is sent as a result of a Control-menu box selection. Most applications do not intercept system messages.

Clipboard messages

These messages are sent when another application attempts to access data in the Clipboard when your application owns or is viewing it.

System information messages

These messages are sent when a Windows system-level attribute, such as color or font, changes.

Control manipulation messages

These messages are sent by an application to its controls, such as list boxes and buttons. Each control class understands its own set of messages for querying and setting its attributes. Control manipulation messages differ from all other messages except MDI messages in that they do not *notify* the application about an event, they *cause* an event. See "Sending messages" on page 98.

Control notification messages

These are WM_COMMAND messages that notify a parent window of a control event, such as a list box item being selected or an edit control being typed into. The message's parameters specify a particular notification code, such as LBN_SELCHANGE and EN_CHANGE.

Scroll-bar notification messages

These are specialized control-notification messages for scroll bar controls. There are two messages, WM_HSCROLL and WM_VSCROLL.

Non-client area messages

These messages, including WM_NCMOUSEMOVE and WM_NCPAINT, are similar to input messages, but deal specifically with events outside a window's client area, including its menu bar, title bar, and borders. Usually your application will not need to redefine the default handling of these messages.

Multiple document interface messages

These messages are sent by an application that follows the MDI standard to manipulate its child windows. MDI messages include WM_MDIACTIVATE and WM_MDIDESTROY.

Default message processing

Windows provides default responses for many of the messages that it sends to an application. When your control, dialog box, or window ignores an incoming message by not defining a corresponding message response member function, ObjectWindows automatically invokes the default processing supplied by Windows. The appropriate default processing for a control, dialog box, or window is specified by its **DefWndProc** member function.

See "Default message processing" in Chapter 9, "Interface objects."

However, when you define a response method for an incoming message, you may want to additionally perform the Windows-supplied response. To do so, call your control, dialog, or window's **DefWndProc** member function from a message response member function.

Sending messages

One source of messages is your program itself. Your program can send messages to itself by calling two Windows API functions, **SendMessage** and **PostMessage**. **SendMessage** forces the window whose handle you pass to process the message immediately while **PostMessage** places the message into the window's application queue and returns without waiting for the message to be processed. This technique is also useful to simulate an event from within your program. More information about these API functions is available in the online Help.

SendMessage is most useful when manipulating controls, such as list boxes and buttons. Windows defines *control-manipulation messages* to do such things as add items to a list box (LB_ADDSTRING) or change the state of a check box (BM_SETCHECK). ObjectWindows's control objects define member functions, such as **TListBox::AddString** and **TCheckBox::SetCheck**, that send many of these messages (those commonly used) to their associated interface elements. If you want to send some of the more obscure messages to controls in your windows, use **SendMessage**, which passes the control's *HWindow* as the first (*Wnd*) parameter.

See "Manipulating dialog controls" on page 145.

When you want to send a message to a control in a dialog box, the procedure is somewhat different. Often, the controls in your **TDialog** objects will not have ObjectWindows interface objects associated with them, but you can send messages to them using **TDialog::SendDlgItemMsg**. You can facilitate communication with controls in your dialog boxes by associating interface objects with them, using an alternate **TWindow** constructor.

Message ranges

Since messages are defined by **WORD**-type values, there are 65,536 possible messages. ObjectWindows divides messages into unique ranges for standard Windows messages, command messages, notification messages, and so on. To make these ranges more identifiable, ObjectWindows defines constants to use as offsets for the separate ranges. The message ranges and their ObjectWindows constants are listed in Table 7.1.

Table 7.1: Ranges of Windows messages

Constant	Value	Message range	Meaning
WM_FIRST	0x0000	0x0000-0x03FF	Windows messages
WM_USER	0x0400	0x0400-0x7F00	Programmer-defined window messages
WM_INTERNAL	0x7F00	0x7F00-0x7FFF	Reserved for internal use
ID_FIRST	0x8000	0x8000-0x8EFF	Programmer-defined child-ID messages
ID_INTERNAL	0x8F00	0x8F00-0x8FFF	Reserved for internal use
NF_FIRST	0x9000	0x9000-0x9EFF	Programmer-defined notification messages
NF_INTERNAL	0x9F00	0x9F00-0x9FFF	Reserved for internal use
CM_FIRST	0xA000	0xA000-0xFEFF	Programmer-defined command messages
CM_INTERNAL	0xFF00	0xFF00-0xFFFF	Reserved for internal use

User-defined messages

Windows lets you define your own messages for use by your application. User-defined messages are useful when defining and responding to a new event. For example, you can have one window send a user-defined message to all the remaining open windows in the application.

Associated with each Windows message is a **WORD**-type identifier. For each of the predefined Windows messages, windows.h defines a corresponding **WORD**-type constant, which is the constant used in the declaration of a message response member function.

Within the allowable range of Windows message identifiers are values reserved for user-defined messages. The range of values reserved for predefined Windows messages is 0 to WM_USER – 1. The allowable range of values for user-defined messages is WM_USER to WM_USER + 31,744. It's probably best to define user-message values as constants, starting with the values WM_USER, WM_USER + 1, WM_USER + 2, and so on. For example,

```
#define WM_MYFIRSTMESSAGE  WM_USER
#define WM_MYSECONDMESSAGE WM_USER + 1
#define WM_MYTHIRDMESSAGE  WM_USER + 2
```

Then send your messages using the Windows function, **SendMessage**:

```
SendMessage(AWindow->HWindow, WM_MYFIRSTMESSAGE, AWord, ALong);
```

To receive a user-defined message, follow the same procedure as for a standard Windows message. For example,

```
class  MyWindow : public TWindow
{
:
  virtual void WMMyFirstMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYFIRSTMESSAGE];
  virtual void WMMySecondMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYSECONDMESSAGE];
  virtual void WMMyThirdMessage(RTMessage Msg) =
    [WM_FIRST + WM_MYTHIRDMESSAGE];
:
};
```

C H A P T E R

8

Module and application objects

The first requirement of an ObjectWindows application is the definition of an application class derived from the ObjectWindows **TApplication** class. The application object will inherit the following behavior of an ObjectWindows application:

- Creating and displaying the application's main window.
- Initializing the first instance of an application for any tasks that serve all instances of the application, such as configuring a communications port.
- Initializing every instance of an application; for example, to load an accelerator table. (You can simultaneously run many instances of the same ObjectWindows application.)
- Processing the Windows messages an application will receive.
- Closing the application.

Besides defining a class derived from **TApplication**, you must add to it the ability to construct the main window object. You also have the option to redefine the default behavior for initializing instances, closing the application, and processing Windows messages.

Application flow

The main program of your ObjectWindows application will normally consist of just three statements.

```
THelloApp HelloApp
("HelloApp", hInstance,
hPrevInstance, lpCmdLine,
nCmdShow);
```

The first statement constructs the application object by calling its constructor. The constructor also initializes the data members of the application object. The application object can be accessed from other parts of an ObjectWindows program by calling **GetApplication**.

```
HelloApp.Run();
```

The second statement calls the application's **Run** member function, which then calls **InitApplication** and **InitInstance** to perform first-instance and each-instance initialization, respectively. **InitMainWindow** is then called to create a main window. In most cases, you will need to redefine only **InitMainWindow** (see Figure 8.1).

Run then sets the application in motion by calling **MessageLoop** to begin processing incoming Windows messages, the instructions that direct an application's activity. **MessageLoop** calls member functions that process particular incoming messages. **MessageLoop** is exactly that, a message loop, which continues running until the application closes.

```
return HelloApp.Status;
```

The third statement returns the final status of the application that ObjectWindows stores in the *Status* data member. This is necessary because **WinMain** must return an **int**. The value of *Status* can also be useful when debugging because a value other than zero indicates an error.

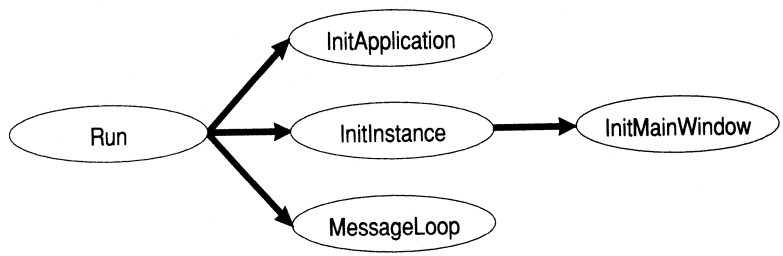


Figure 8.1
Member function calls that control an application's flow

Initializing applications

The flow of member function calls that initialize the application object let you customize many parts of the process by redefining member functions. The one required member function you must write is **InitMainWindow**. You can also redefine **InitInstance** to perform special initialization for each executing instance of an

ObjectWindows application, and **InitApplication** to initialize the first executing instance.

Initializing the main window

Here is a minimal application class definition:

This code fragment comes from HELLOAPP.CPP.

```
class THelloApp :public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
            hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};
```

You must define an **InitMainWindow** member function, which constructs the main window object and stores it in the application object's *MainWindow* data member. Here is a sample **InitMainWindow** member function:

```
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}
```

This simple ObjectWindows application consists of a main program plus definition of a class derived from **TApplication** that defines one member function, **InitMainWindow**:

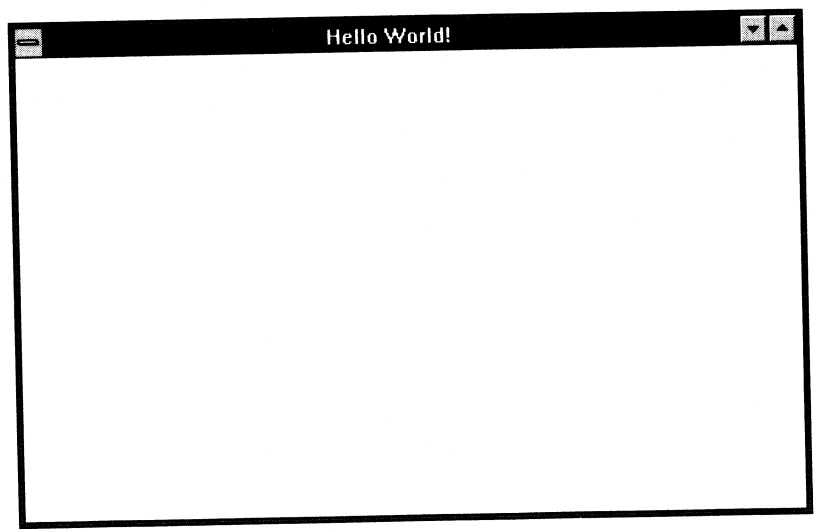
This is HELLOAPP.CPP.

```
#include <owl.h>

// Define a class derived from TApplication
class THelloApp :public TApplication
{
public:
  THelloApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
            hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

// Construct the THelloApp's MainWindow data member
void THelloApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, "Hello World!");
}
```

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  THelloApp HelloApp ("HelloApp", hInstance, hPrevInstance,
                      lpCmdLine, nCmdShow);
  HelloApp.Run();
  return HelloApp.Status;
}
```

The preceding program is a minimal application that simply displays a window with the caption "Hello World!", as shown in Figure 8.2. You can move and resize this window. You can also minimize it to an icon by clicking the ↓ icon in the upper right corner. To restore it, double-click the icon. Clicking the ↑ icon maximizes it to full screen. To close the window and terminate the application, double-click the Control-menu box in the upper left corner.

Figure 8.2
The Main Window



Initializing each application instance

The **InitInstance** member function of an application class performs initialization for each instance of a Windows application. **TApplication** defines an **InitInstance** member function that calls **InitMainWindow** and then displays the main window with a call to its **Show** member function. You can redefine **InitInstance** to modify the standard initialization; for example, to load an accelerator table, an application-oriented activity. If you redefine **Init-**

InitInstance for your application class, be sure it calls **TApplication::InitInstance** first.

Here is a sample **InitInstance** member function that loads an accelerator table before the application is set in motion. The resource identifier of the accelerator table is 100, which is defined in the resource file.

```
void TMyApp::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(GetApplication()->hInstance,
                               MAKEINTRESOURCE(100));
}
```

InitInstance should only perform initialization of the application instance. Main window initialization should be done in the constructor and in the **SetupWindow** member function of the main window object.

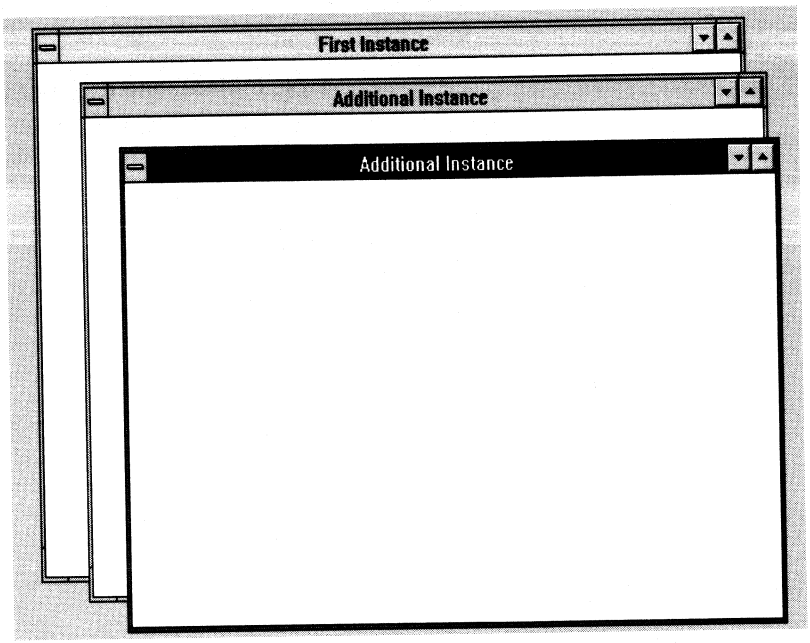
Initializing the first application instance

If the user runs your application many times, simultaneously (without terminating it), you can define some processing to occur the first time it is run. This processing is called the first-instance initialization. Note that if the user starts and terminates your application, starts it again, and so on, each instance is considered a first instance.

If the current instance is a first instance, its **InitApplication** member function will be called. **TApplication** defines a "do-nothing" **InitApplication** member function that can be redefined to perform special first-instance initialization.

For example, you could modify **TestApp** to make the main window's caption reflect whether or not it is the first instance. To do this, add a *WindowTitle* data member to the application class. Then define an **InitApplication** member function that sets *WindowTitle* to "First Instance." Make the application constructor initialize *WindowTitle* to "Additional Instance." **InitApplication** is called only for the first instance.

Figure 8.3
Refining application initialization



This is INSTTEST.CPP.

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

class TTestApp : public TApplication
{
public:
  TTestApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
           lpCmdLine, int nCmdShow);

protected:
  char WindowTitle[20];
  virtual void InitMainWindow();
  virtual void InitApplication();
};

TTestApp::TTestApp(LPSTR AName, HANDLE hInstance, HANDLE
                   hPrevInstance, LPSTR lpCmdLine, int nCmdShow) :
                   TApplication(AName, hInstance, hPrevInstance,
                   lpCmdLine, nCmdShow)
{
  strcpy(WindowTitle, "Additional Instance");
}

void TTestApp::InitApplication()
{
  strcpy(WindowTitle, "First Instance");
```

```
}

void TTestApp::InitMainWindow()
{
  MainWindow = new TWindow(NULL, WindowTitle);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TTestApp TestApp("Instance Tester", hInstance, hPrevInstance,
                   lpCmdLine, nCmdShow);
  TestApp.Run();
  return TestApp.Status;
}
```

Running applications

Your application's message loop is invoked when your program calls its application object's **Run** member function, which calls its **MessageLoop** member function. Throughout the operation of your program, the message loop processes incoming Windows messages. Your ObjectWindows programs will inherit a **MessageLoop** member function that works automatically.

The **MessageLoop** member function calls three translation member functions to process Windows messages. **ProcessDlgMsg** handles modeless dialog messages, **ProcessAccels** handles accelerators, and **ProcessMDIAccels** handles accelerators for MDI applications. For applications that do not use accelerators or modeless dialog boxes, or which are not MDI applications, you may want to streamline the **MessageLoop** somewhat. See the entries for the translation member functions under **TApplication** in Chapter 16, "Class reference."

Closing applications

Before your application exits, the program must break out of the message loop. This may happen as a result of the user attempting to close the main window of the application. We say it *may* happen because ObjectWindows provides a mechanism to redefine an application's closing behavior; for example, to check for unsaved files before closing.

Here's what happens when the user tries to close an application by double-clicking its Control-menu box:

1. Windows sends a WM_CLOSE message to the application's main window.
2. The main window object responds by calling its **CloseWindow** member function, which closes the main window (by calling **ShutDownWindow**) only after checking with the application object through a call to its **CanClose** member function (because closing the main window of the application amounts to closing the application).
3. The application object's **CanClose** member function returns the result of a call to the **CanClose** member function of the main window, allowing the main window itself to determine whether it should be closed.
4. The main window, by default, decides if it's OK to close itself by asking its child windows if it's OK to close. If all the child windows' **CanClose** member functions return TRUE, the main window's **CanClose** returns TRUE, and the application closes.

This mechanism is comparable to announcing, "If there's anyone who knows why this application should not close, please speak now or forever hold your peace." In the end, the application object has the responsibility for giving the approval for closing the application. By default, it checks with the main window before giving this approval. You can also redefine the **CanClose** member function of the main window class, which will result in some other closing behavior. You are also free to redefine the **CanClose** member function of the application class if you'd like to change the default behavior.

C H A P T E R

9

Interface objects

Objects representing windows, dialog boxes, and controls are called user interface objects, or simply interface objects. This chapter discusses the general requirements and behavior of interface objects and their relationship with the actual windows, dialog boxes, and controls that appear on the screen.

This chapter also explains the relationship between the different interface objects of an application, as well as the mechanism for responding to Windows messages.

TWindowsObject

TWindowsObject is a base class common to all Windows interface classes: **TDialog**, **TWindow**, and its derived class, **TControl**. **TWindowsObject** defines behavior common to window, dialog, and control objects. **TWindowsObject** member functions do the following:

- Maintain the dual ObjectWindows object/Windows element structure, including constructing and destroying objects, and creating and destroying elements.
- Automatically support parent-child relationships between interface objects (not class inheritance relationships).
- Register new Windows classes; see Chapter 10, "Window objects."

The work of **TWindowsObject** is behind the scenes. You will rarely, if ever, derive new classes directly from **TWindowsObject**. It defines much of the functionality an object inherits when you derive new classes from **TWindow** and **TDialog**.

Why interface objects?

Why do you need interface objects if Windows already has visual windows, dialogs, and controls?

Each interface object has an associated *visual interface element*—not an object, but a physical window, dialog, or control—for which it acts as an object-oriented surrogate. The interface object provides member functions for creating, initializing, managing, and destroying its associated interface element. Data members of the interface object hold associated data, including its interface element's handle and its parent and child windows. The interface object's member functions handle many of the details of Windows programming for you.

The object/element relationship is a lot like that between a C++ stream and a DOS file. You can construct a stream object to handle the details of DOS file I/O. You construct a window object to handle the details of window I/O.

The structure of an interface object, with an associated interface element, requires some awareness on your part to properly manipulate windows, dialogs, and controls. For example, to create a complete interface object, you have to call *two* member functions. The first is a constructor, which constructs the interface object and sets any of its attributes, such as style and menu.

The second member function is the creation member function, **MakeWindow**, which associates the interface object with a new interface element. This association is maintained by the interface object's *HWindow* data member, which holds a handle to a window. **MakeWindow** is a member function of the application object. **MakeWindow** calls the object's **Create** member function, which tells Windows to create a visual element.

MakeWindow "safely" creates a visual element. **MakeWindow**, unlike **Create**, ensures that the interface object has been properly constructed and that there is sufficient memory. Any speed overhead involved in using **MakeWindow** is well worth making sure you won't run out of memory and crash Windows.

Similarly, the visual interface element must be *destroyed* and the corresponding interface object must be *freed*. However, here you need only call **ShutDownWindow**, which destroys the interface element and then deletes the interface object.

You should note that creating both an interface object and corresponding visual element doesn't necessarily mean you'll see something onscreen. When creating the visual element, Windows checks to see if the window has the style WS_VISIBLE set. WS_VISIBLE and other window styles are set or cleared in the constructor, by setting or clearing them in the interface object's **Attr.Style** data member. The interface element is displayed when created only if WS_VISIBLE is set.

At any point, an element can be shown or hidden by calling the interface object's **Show** member function.

Window parent-child relationship

In a Windows application, interface elements (windows, dialog boxes, and controls) are related through parent-child links. Two interface elements are related when one is the parent window of the other, the child window. Don't confuse this parental relationship with inheritance or with instance ownership, which are both object relationships. A child window doesn't necessarily descend from its parent window or inherit anything from it.

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 spawn them. When you close the parent window, the child windows automatically close; similarly, if you move the parent window, the child windows move along with it.

All interface elements (windows, dialogs, and controls) can serve as parent or child windows.

Child window lists

You specify an interface element's parent at the time you construct it. The parent window object is a parameter of the interface object's constructor. A child window object keeps track of its

parent window object by storing a pointer to the object in its *Parent* data member. It also keeps track of *its* child window objects automatically.

When you define a new interface class with child windows, define the constructor to also construct the child window objects. Then, when the application calls the parent window's **Create** member function, the parent's interface element is created. If successful, the **SetupWindow** member function is called in the process. **TWindowsObject::SetupWindow** iterates over every child window in its private list of children, calling the **Create** member function for each one except dialogs, by default. Child windows with the WS_VISIBLE state set are displayed as soon as they are created.

Often you will redefine the **SetupWindow** of the parent window to perform some setting-up tasks after its child windows are created. Filling a list box with elements is one example. In that case, be sure to invoke the **SetupWindow** member function of your window object's base class as the first statement in your redefined **SetupWindow** member function.

See Step 10 on page 71 for an example of redefining SetupWindow.

To explicitly exclude a child window (such as a pop-up window) from this default handling, call the member function **DisableAutoCreate** after constructing the object. To explicitly request default handling for a child window (such as a dialog box, which would normally be excluded), call its **EnableAutoCreate** member function.

Just as calling the parent window's **Create** member function results in calling its child windows' **Create** member functions, calling the parent's destructor results in calling the destructors of all its child windows. The **CanClose** member function also iterates over its private list of children calling each child window object's **CanClose** member function, returning TRUE only if all child windows also return TRUE.

Child window iterators

You may want to write member functions that iterate over each of a window's child windows to perform a function. For example, you might want to check all the child check boxes in a window. In that case, use the **ForEach** member function inherited from **TWindowsObject**, as shown in the following example.

The **ForEach** and **FirstThat** member functions expect a pointer to a function as their first argument. See the ObjectWindows class reference for more information on **TWindowsObject::ForEach** and **TWindowsObject::FirstThat**.

```
void CheckTheBox(Pvoid P, Pvoid Param)
{
  (PTCheckBox)P->Check();
}

void TMyWindow::CheckAllBoxes();
{
  ForEach(CheckTheBox, NULL);
}
```

You may also want to write member functions that iterate over a window's list of child windows looking for a particular child. For example, in a window with many check box child windows, you might want to find the first check box that is checked. To do so, use the **FirstThat** member function inherited from **TWindows-Object** as follows:

```
BOOL IsThisBoxChecked(Pvoid P, Pvoid Param)
{
  return ((PTCheckBox)P->GetCheck() == BF_CHECKED);
}

PTCheckBox TMyWindow::GetFirstChecked();
{
  return (FirstThat(IsThisBoxChecked, NULL))
}
```

Message processing

ObjectWindows programs have a two-way communications channel with Windows. In one direction, the application controls the user interface by calling Windows functions. In the other

direction, Windows sends messages back to the application in response to *events* in the program, such as the user choosing a menu selection (WM_COMMAND) or clicking the left mouse button (WM_LBUTTONDOWN). There are over 100 standard Windows messages; you can also create your own *user-defined* messages, as explained in Chapter 7, "Overview."

A Windows application performs processing in response to messages received from Windows. Application-specific behavior can be achieved by selectively responding to the Windows messages received. Windows functions can be called in the process, and can themselves generate new Windows messages.

Incoming messages are always intended for a specific window—the window the current event most directly affects. For example, when you click the left mouse button in a particular window, the WM_LBUTTONDOWN message generated is intended for that window. ObjectWindows maps the incoming messages for your windows to their response member functions.

This interplay of Windows messages and object member functions forms the skeleton of an ObjectWindows application. As you can see, the main duty of an ObjectWindows application is to respond to Windows events. This is a very different orientation than the traditional program-controlled DOS application. This is message-based, event-driven programming. You can think of it as *not speaking until you are spoken to*. It requires discipline, but it pays off in well-behaved, multitasking applications.

Responding to messages

See page 9 for more information about message response member functions.

In order to design your window objects to respond to incoming Windows messages, you must write member functions that correspond, on a one-to-one basis, to each incoming message you care to respond to. These member functions are called *message-response member functions*. This special message-to-member function correspondence is facilitated by an extension to the virtual function declaration. To mark a member function as a message-response member function, add the sum of WM_FIRST and the message name constant to the end of the virtual member function declaration:

```
class TMyWindow : public TWindow {
public:
  virtual void WMRButtonDown(RTMessage Msg) =
    [WM_FIRST + WM_RBUTTONDOWN];
};
```

In this example, the **WMRButtonDown** member function is invoked when a **TMyWindow** object gets a WM_RBUTTONDOWN message. The member function and message names need not be similar, but it can improve the readability of your programs. ObjectWindows uses this convention in its interface classes.

Msg is a required argument of type **RTMessage**. **RTMessage** is an ObjectWindows-defined type that is a reference to a **TMessage** structure containing *WParam*, *LParam*, and *Result* fields. The *WParam* and *LParam* members (of type **WORD** and **LONG**) contain information about the event that occurred, prompting the message. Often, each parameter holds two pieces of information. For example, when WM_LBUTTONDOWN is sent, the low-order **WORD** of *LParam* holds the x-coordinate of the clicked point, and the high-order **WORD** holds the y-coordinate.

LParam is declared as a member of a union along with an *LP* structure, which declares *Hi* and *Lo* members of type **WORD**. You can therefore use *Msg.LP.Hi* and *Msg.LP.Lo*. to refer to the high- and low-order words of *Msg.LParam*. *WParam* is similarly declared as a member of a union along with a *WP* structure. Most often, you will use the *WParam*, *LP.Lo*, and *LP.Hi* fields, all **WORD** types. *Msg* is passed by reference because a few Windows messages require a response that your member functions will store in its *Result* member.

In your ObjectWindows programs, you can safely ignore Windows messages to which you do not want your objects to respond. You do this by not defining member functions that respond to those messages. Those messages will be handled by default by the **DefWndProc** member function your window inherits **TWindowObject**.

There are shortcuts to the ObjectWindows message-response model for a few common messages associated with menus and child windows, such as edit controls and scroll bars, as outlined in the following sections.

Command and child window messages

Windows generates a *command-based message* when the user selects a menu choice or types an accelerator. It generates a *child-ID-based message* when the user activates a control, such as a button or a list box. Most of these actions result in a WM_COMMAND message.

To help sort out all the possibilities and avoid a lengthy case statement, ObjectWindows automatically responds to WM_COMMAND messages by generating another, more specialized, virtual member function call based on the contents of the message. You define *command-based* and *child-ID-based* message-response member functions using a member function declaration extension, similar to the one already described.

Command message processing

To associate an event generated by a menu or accelerator with a command message-response member function, use an identifier which is the sum of the ObjectWindows-defined constant, CM_FIRST and the menu or accelerator ID defined in the menu or accelerator resource. For example, take a window, a **TFileWindow** object, that has three menu options, File | New, File | Open and File | Save. The menu IDs defined in owlrc.h are CM_FILENEW, CM_FILEOPEN, and CM_FILESAVE, respectively. Here's how the **TFileWindow** object definition might look:

```
class TFileWindow : public TWindow
{
:
  virtual void CMFileNew(RTMessage Msg) =
    [CM_FIRST + CM_FILENEW];
  virtual void CMFileOpen(RTMessage Msg) =
    [CM_FIRST + CM_FILEOPEN];
  virtual void CMFileSave(RTMessage Msg) =
    [CM_FIRST + CM_FILESAVE];
:
}
```

However, a window that owns a menu doesn't always have to define a response to a command-based message. One of its child windows, including its controls, can define a response instead. The child window that has the input focus at the time of the menu or accelerator selection gets the first opportunity to respond to a command-based message by defining a response member function. If a window does not define a response, the message is sent

to that child window's parent window, and to that window's parent, and so on, until it finds a response or reaches the window that owns the menu. This means that an edit control in a window can respond to editing menu selections directly. See ObjectWindows **TEditWindow** class as an example of this technique.

Child message processing

When the user affects a control (a child window), such as clicking a button or typing in an edit control, its parent window or dialog object receives a *notification message* from the child. To have the parent respond to notification messages, define a member function with an identifier that is the sum of the ObjectWindows-defined constant ID_FIRST and the child window ID defined in a dialog resource or in the construction of a control object. For example, consider a window with a list box. The following code defines a member function called **HandleListBoxMsg** to respond to the child-ID-based messages generated by the list box:

```
class TListBoxWindow : public TWindow
{
⋮
  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
}
```

where ID_LISTBOX is a constant defined to equal the control ID. Child window IDs must be positive integers smaller than 4,096.

As an alternative to having the parent window respond to child-event messages, you can have the control define a *notify-based* response member function. See "Responding to control notification messages" in Chapter 12, "Control objects."

Default message processing

For more information about notify-based messages, see Chapter 12, "Control objects."

This section applies to Windows messages you intercept directly, using the WM_FIRST constant (that is, Windows messages), but not to command-based, notify-based, or child-ID-based messages.

A Windows application receives many more messages than it needs to respond to. Windows supplies default responses for many of these messages, particularly those sent to controls. For example, when a user types into an edit control, Windows' default response is to display the new characters, scrolling when appropriate. Windows default responses are invoked for an ObjectWindows window objects (including dialog and control objects) by its **DefWndProc** member function.

When you ignore a Windows message (by not defining a member function to respond to it), ObjectWindows automatically invokes the interface object's **DefWndProc** member function. However, even if you intercept a message by defining a message-response member function, you still might need to explicitly call **DefWndProc**.

For example, the following code intercepts the WM_CHAR message, which is sent to an edit control object when the user types into an edit control. Here is the WM_CHAR message-response member function for the class derived from **TEdit**:

```
void TMyEdit::WMChar(RTMessage Msg)
{
  MessageBeep(0);
}
```

This member function responds by sounding a beep as the user types. However, defining a response prohibits the standard response defined by the default window procedure. Thus, you hear the beep, but the text in the edit control is not changed.

What you probably wanted to do, however, was to sound the beep *and* elicit the standard response. To do this, you can explicitly call the **DefWndProc** member function:

```
void TMyEdit::WMChar(RTMessage Msg)
{
  MessageBeep(0);
  DefWndProc(Msg);
}
```

In the previously described member function, the beep is sounded and then the character is added to the edit control.

In general, you call **DefWndProc** under these circumstances:

- When you want to be aware that a message was received, but do not want to override the default message processing.
- When you intercept a message to perform some action other than the standard response, but you also want the standard response.
- When intercepting a message seems to disable some expected functionality.

C H A P T E R

10

Window objects

This chapter explains how to create, display, and fill an application's windows. The ObjectWindows class **TWindow** defines most of the fundamental behavior of the program's main window and any of its windows. **TWindow** defines the behavior for opening, closing, painting, and scrolling windows, as well as for child window and command message processing.

TWindow has four derived classes: **TMDIFrame**, **TMDIClient**, **TControl**, and **TEditWindow**. **TMDIFrame** and **TMDIClient** are used in ObjectWindows applications that conform to the Windows multiple document interface (MDI) standard. For an explanation of MDI and these types, see Chapter 14. **TControl** defines controls, such as buttons and list boxes, and is covered in Chapter 12. **TFileWindow**, described herein, derives from **TEditWindow**. Most often, you will create new window classes derived from **TWindow**.

This chapter covers **TWindow**, **TEditWindow**, and **TFileWindow**, and includes an example of registering a new Windows class.

The TWindow class

At a minimum, an ObjectWindows application must have a main window that it displays when the application starts up. An ObjectWindows program's main window is usually an instance of a class derived from **TWindow**. Many applications have other

windows that are usually child windows of the main window. These additional windows are also usually instances of a class derived from **TWindow**. For example, a painting program might define **TPaintWindow** for its main window, and **TZoomWindow** for another window that shows an enlarged view of the painting. In this case, both **TPaintWindow** and **TZoomWindow** would descend from **TWindow**. In a well-designed application, **TZoomWindow** would probably be a specialized version of **TPaintWindow** and implemented as a class derived from **TPaintWindow** (and **TWindow**, indirectly).

Initializing and creating window objects

Like dialog and control objects, window objects are interface objects that are associated with visual interface elements. More accurately, they represent window elements, identified through a handle stored in their *HWindow* data member. Since a window has two parts, creating one takes two steps: constructing the object, and creating the visual element, in that order.

Initializing window objects

A typical Windows application has many different styles of windows: overlapped or pop-up, bordered, scrollable, and captioned, to name a few. These style attributes, as well as other creation attributes, are usually set when a window object is constructed and used when the visual element it represents is created.

A window object's creation attributes, such as its style, title, and menu, are stored in its *Attr* data member, which is defined as type **TWindowAttr**. A **TWindowAttr** includes the following data members:

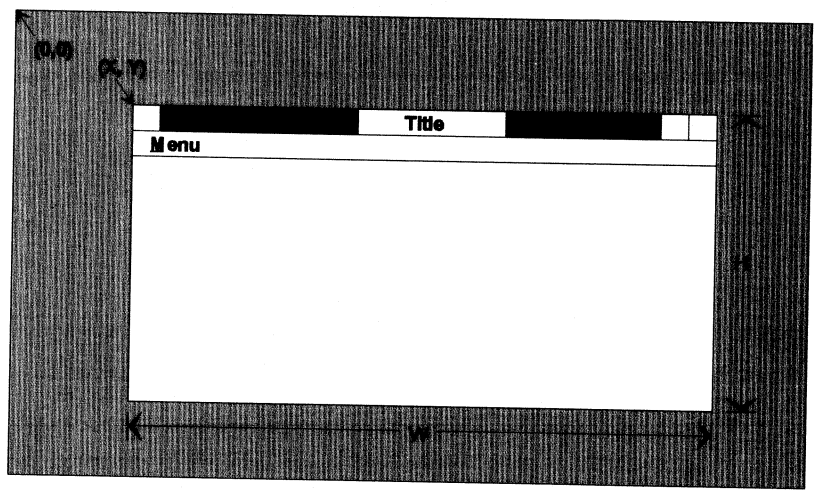
Table 10.1
TWindowAttr data members

Data member	Use
Style	A **DWORD** that holds the combined style constant.
ExStyle	A **DWORD** that holds the extended style.
Menu	An LPSTR that identifies the menu resource.
X	An **int** representing the window's initial horizontal location. It is the horizontal screen coordinate of the window's upper left corner.
Y	An **int** representing the window's initial vertical location. It is the vertical screen coordinate of the window's upper left corner.

Table 10.1: TWindowAttr data members (continued)

W	An **int** representing the window's initial width, in screen coordinates.
H	An **int** representing the window's initial height, in screen coordinates.
Param	A **LPSTR** that will be passed to the window when it is created.
Id	An **int** specifying the child window ID used for communication between a control and its parent window or dialog. *Id* should be unique for all child windows of the same parent. If the control is defined in a resource then its *Id* should be the same as the resource ID. A window should never have both *Menu* and *Id* set.

Figure 10.1
A window's attributes



The first argument in the constructor call is the window's parent window object. It is NULL if the window is the main (parentless) window. **TWindow**'s constructor sets the *Title* data member to the specified **LPSTR** argument. As a default, it also sets the *Style* data member of *Attr* to WS_OVERLAPPEDWINDOW if the window is the application's main window (WS_VISIBLE otherwise). *X* and *W* are set to CW_USEDEFAULT and *Y* and *H* are set to 0, which results in a reasonably-sized overlapped window. This is the standard way to size a new main window.

When you create window classes that derive from TWindow you must define a constructor that, at a minimum, calls its base class's constructor:

```
class MyWindow : public TWindow
{
:
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle) : TWindow(AParent,
            ATitle) {}
:
};
```

You can then reset your new window object's attributes by directly modifying its *Attr* data member in its constructor. For example, you might want the window to come up maximized:

```
class TMyWindow : public TWindow
{
:
  TMyWindow(PTWindowsObject AParent, LPSTR ATitle);
:
  PTChildWindow AChildWindow;
  PTListBox ListBox;
};
:
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                     TWindow(AParent, ATitle)
{
  Attr.Style |= WS_MAXIMIZE;
  AChildWindow = new TChildWindow(this, "Child Title");
  ListBox = new TListBox(this, ID_LISTBOX, 201, 20, 20, 180, 80);
:
}
```

Notice, in the previous example, that the sample window's constructor is responsible for constructing its child window objects, such as pop-up windows and list boxes. In turn, a child window can reset its attributes in its own constructor:

```
TChildWindow::TChildWindow(PTWindowsObject AParent, LPSTR ATitle) :
                           TWindow(AParent, ATitle)
{
  Attr.Style |= WS_POPUP | WS_CAPTION;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 300;
  Attr.H = 300;
}
```

For a complete list, see the online Help.

The *Style* attribute can be one or a combination of window style constants, such as

- WS_OVERLAPPEDWINDOW

- WS_POPUPWINDOW
- WS_CHILDWINDOW
- WS_CAPTION
- WS_BORDER
- WS_VSCROLL

As an alternative, if you choose not to define a derived class for a child window, you can construct the window object, and then reset its attributes, all from within the parent window's constructor:

Note that you can't directly set the Attr.Menu member; use the AssignMenu member function instead.

```
TMyWindow::TMyWindow(PTWindowsObject AParent, LPSTR ATitle) :
                     TWindow(AParent, ATitle)
{
  AssignMenu(100);
  AChildWindow = new TWindow(this, "Child Title");
  AChildWindow->Attr.Style = WS_POPUP | WS_CAPTION;
  AChildWindow->Attr.X = 100;
  AChildWindow->Attr.Y = 100;
  AChildWindow->Attr.W = 300;
  AChildWindow->Attr.H = 300;
  :
}
```

Creating window elements

Creating a window element is the process of building the visual elements associated with a window object. You do this by calling the application object's **MakeWindow** member function, passing the window object as a parameter:

GetApplication() returns a pointer to the application object.

```
if (GetApplication() -> MakeWindow(AWindow))
  // creation successful
else
  // creation unsuccessful
```

MakeWindow calls two important member functions. The first is **ValidWindow**, which checks whether the window object was constructed successfully. If the constructor failed for any reason, **MakeWindow** deletes the window object and returns NULL. If the constructor was successful, **MakeWindow** goes on to call the window object's **Create** member function. **Create** is the member function that actually tells Windows to create a visual element. If **Create** fails, **MakeWindow** deletes the window object and returns NULL. Otherwise, it returns a pointer to the window object.

Although it is the member function that actually creates the window element, you won't normally call **Create** explicitly. The application's main window is automatically created upon application startup by **TApplication::InitInstance**. All other application windows are child windows, either directly or indirectly, of the main window, and child windows are usually either created in the **SetupWindow** member function of their parent window objects. Windows can be created dynamically, "on the fly," by calling **MakeWindow**.

Initialization and creation summary

When designing a new window class, you can define its constructor to set the window's attributes and construct any child window objects. (A window's attributes can also be set in the parent window's constructor.)

GetApplication()->MakeWindow creates a window object's visual element by calling the window's **Create** member function. If WS_VISIBLE is among the styles set in its *Attr.Style* data member, the window will be displayed when it is created. Afterwards, the window's **SetupWindow** member function is called, which by default creates its child windows.

The parent window's constructor usually constructs its child windows. A window object's attributes are usually set by its constructor or by the constructor of its parent window. Since an application's main window has no parent, the application object constructs and creates it upon application startup.

Window class registration

You can update a window's menu by calling the AssignMenu member function.

You learned that during window object initialization, you can set attributes of a window, such as its style and location, by filling the object's *Attr* data member. Because these attributes are used in creating the corresponding window element, they are called *creation attributes*.

There are also other attributes, including background color, representative icon, and mouse cursor. To see some of the other attributes, run Windows with a few different types of applications at once. As you move the mouse cursor from one application to another, the cursor may change from an arrow (regular) to a wide cross (for spreadsheets) to a thin cross (for graphics programs).

These "other" attributes of a window (including its icon) are specified by identifying the *Windows class* of the window when the window is created. The attributes defined for a Windows class, unlike creation attributes, cannot be changed on a window-by-window basis but hold for all windows with the same Windows class.

The Windows class of the window is not the same as the class of the ObjectWindows window object. Each instance of an ObjectWindows window class does, however, share the same Windows class.

A Windows class must be *registered* with Windows before a window with that Windows class is created. ObjectWindows takes care of the actual registration for you when your windows are created. ObjectWindows also supplies a default Windows class with default *registration attributes*. If you don't want to change any of these defaults, you don't have to worry about registration at all.

If you change any of these attributes for one of your windows, you'll need to associate the window with a new Windows class that you define. To do so, you'll redefine two virtual member functions of your window: **GetWindowClass** and **GetClassName**. In **GetWindowClass** you'll identify the registration attributes you want. You'll also supply a name for the Windows class by redefining **GetClassName**. **GetClassName** is a function that simply returns the name (**LPSTR**) of the window class. **TWindows** defines a **GetClassName** member function that returns "OWLWindow," the name of the default window class.

To define a window class, called **IBeamWindow**, that uses an I-beam cursor rather than the standard arrow, redefine the inherited **GetClassName** member function as follows:

```
LPSTR IBeamWindow::GetClassName()
{
  return "IBeamWindow";
}
```

The returned class name is used as an identifier for the Windows class when it is registered with Windows.

The returned class name may or may not be the same as the actual class name. It makes no difference.

The **GetWindowClass** member function takes a **WNDCLASS** structure as a reference argument and fills its fields with new registration attributes. You should first call the **GetWindowClass** member function of your base class, then set the fields you want

to change. In the following example, **TWindow::GetWindowClass** is called and then the *hCursor* field, which stores a handle to a cursor resource, is reset.

```
void IBeamWindow::GetWindowClass(WNDCLASS& AWndClass)
{
  TWindow::GetWindowClass(AWndClass);
  AWndClass.hCursor = LoadCursor(NULL, IDC_IBEAM);
}
```

IDC_BEAM is a constant representing one of Windows' stock cursors. Figure 10.2 shows an application that uses the **IBeamWindow** object.

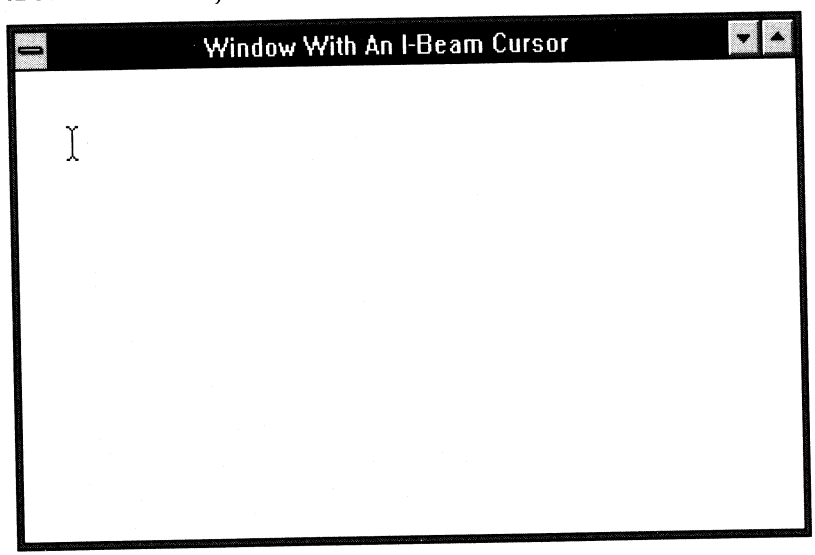


Figure 10.2
A program that uses the IBeamWindow class

Registration attributes

By default, ObjectWindows defines a Windows class that it uses for instances of **TWindow**. "OWLWindow" registration attributes include: a blank icon, an arrow cursor, and standard window color. To change these characteristics for your **TWindow**-derived class, you must redefine its **GetWindowClass** member function. **GetWindowClass** is passed a reference to a **WNDCLASS** structure that you must fill with registration attributes.

Here is a table showing some of the characteristic fields of a **WNDCLASS** structure and the default values to which they are set in **TWindow::GetWindowClass**.

Table 10.2 Window registration attributes

Characteristic	Field	Default
Class style	*Style*	CS_HREDRAW | CS_VREDRAW
Icon	*hIcon*	LoadIcon(0, IDI_APPLICATION)
Cursor	*hCursor*	LoadCursor(0, IDC_ARROW)
Background color	*hbrBackground*	(HBRUSH)(COLOR_WINDOW +1)
Default menu	*lpszMenuName*	NULL

Class style member

The *Style* member of a **WNDCLASS** structure differs from the window style that is specified as a creation attribute of a window object in *Attr.Style*. *Attr.Style* is used to specify a window's visual appearance when it is created, and is set using WS_ ("window-style") constants.

The *Style* member of a **WNDCLASS** structure is used to specify certain behaviors of a window after it is created, and is set using CS_ ("class-style") constants. For example, when CS_HREDRAW is set, the entire window is redrawn when its horizontal size changes. When CS_NOCLOSE is set, the Close option is disabled on the Control menu. This *Style* member can be set to one or a combination of CS_ constants. For a complete list of CS_ constants, see the online Help.

TWindow::GetWindowClass sets this member as follows:

```
AWndClass.style = CS_HREDRAW | CS_VREDRAW;
```

Icon member

The *hIcon* member holds a handle to an icon that is used to represent a window in its minimized state. Usually, you will define an icon resource to represent the main window of your program. **TWindow::GetWindowClass** sets this member to the handle of a stock icon, IDI_APPLICATION, which appears as a blank rectangle.

Cursor member

The *hCursor* data member holds a handle to a cursor that is used to represent the mouse pointer when it is positioned over the window. **TWindow::GetWindowClass** sets this member to the handle of the standard Windows arrow, IDC_ARROW. Some other stock cursors include the following:

- IDC_IBEAM
- IDC_WAIT
- IDC_CROSS

Cursors, like icons, can be user-defined.

Background color member

This field specifies the background color of the window. **TWindow::GetWindowClass** sets this member to the default system window color, which can be set through the Control Panel. You can set this member to the handle of a physical brush of a particular color. Alternatively, you can also set it to any of the system colors, such as COLOR_ACTIVECAPTION by setting *hbrBackground* to COLOR_ACTIVECAPTION + 1.

Default menu member

This field contains a pointer to the resource name of a menu that is shared by all windows of the Windows class. For example, if you were to define an **EditWindow** type that always has a standard editing menu, you could specify that menu here. This would eliminate the need to specify the menu as a creation attribute in a constructor. If this menu resource has a resource ID of 101, you would set this field with

```
AWndClass.lpszMenuName = MAKEINTRESOURCE(101);
```

Scrolling windows

There's more to the world than can be seen through a window. Fortunately, ObjectWindows windows can be easily programmed to scroll their world for the viewer. A **TWindow** enlists the aid of a **TScroller** object to accomplish this feat.

Users normally manipulate scroll bars on the edge of a Window's client area to scroll the current view. (These window scroll bars are not controls, but are part of the window itself, created for windows whose creation styles include WS_VSCROLL or WS_HSCROLL.) A **TScroller** gives life to these window scroll bars, providing an automated way to scroll.

In addition, an ObjectWindows window can also be scrolled when a user drags the mouse from the inside to the outside of the window's client area. We'll call this technique *auto-scrolling;* it works for windows that don't even have scroll bars.

Scroller attributes

A **TScroller** holds values that determine how much of a window is to be scrolled. These values are stored in the **TScroller** data members *XUnit*, *YUnit*, *XLine*, *YLine*, *XPage*, *YPage*, *XRange*, and *YRange*. Data members that start with *X* represent horizontal values, and those that start with *Y* represent vertical values.

A scrolling unit determines the *granularity* of scrolling, which is the smallest number of device units (usually, pixels in a window, but it depends on the present mapping mode) that you can scroll in either the horizontal or vertical direction. The unit is usually based on the kind of information you're displaying. For example, if you are displaying text with an average character width of 8 pixels and a height of 15, then useful values for *XUnit* and *YUnit* would be 8 and 15, respectively.

The other scroller attributes—line, page, and range—are expressed in terms of scrolling units. Line and page values are the number of units to scroll in response to a user's scrolling request. A request initiated by clicking either of the arrows at the end of a scroll bar scrolls the window "a line's worth," the number of units stored in the line data members. A click in the scroll bar itself (but not on the scroll button, or "thumb") scrolls "a page's worth." The range attributes represent the total number of units that can scroll, usually based on the size of the window's universe, such as the document being edited.

As an example, let's look at a text editing window. If you want to display a text file that has 400 lines of text with a limit of 80 characters per line and 50 lines per page, you might choose these values:

Table 10.3
Typical editing window data member settings

Data member	Value	Meaning
XUnit	8	Character width
YUnit	15	Character height
XLine, YLine	1	1 unit per line
XPage	40	40 characters per horizontal page
YPage	50	50 lines per vertical page
XRange	80	Maximum horizontal range
YRange	400	Maximum vertical range

A **TScroller** object with these values allows scrolling of one line or page at a time. The entire file can be viewed by using the scroll bars or by auto-scrolling.

The default line values are 1, so it's not necessary to set these unless something else is desired. There is also a default scheme for setting the page units, based on the current size of the window, so that scrolling a "page" will actually scroll the current client area's height or width, depending on the scrolling direction. It isn't necessary to set the page values yourself unless you want to redefine this mechanism.

Giving your window a scroller

To give your window a scroller, construct a **TScroller** object in your window's constructor and store a pointer to the object in the window data member *Scroller*. Although doing this, you can set the size of the units and the range if known. While window scroll bars are not required for use of a scroller, as in auto-scrolling, many scrollable windows have them. To add them to a window, simply add the window's **Attr.Style** data member in its constructor to WS_VSCROLL, WS_HSCROLL, or both. Here is a constructor for the text editing window example:

This code fragment comes from SCROLAPP.CPP.

```
TScrollWindow::TScrollWindow(LPSTR ATitle) : TWindow(NULL, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 60);
}
```

TScroller's constructor takes, as arguments, the scrollable window and the initial values for the data members *XUnit*, *YUnit*, *XRange*, and *YRange*, respectively. The line and page attributes are set to their default values.

Once this window is displayed, the contents of its client area can be scrolled vertically or horizontally by using the scroll bars or by auto-scrolling. The window's **Paint** member function simply draws the graphical information without needing to know whether the window has been scrolled. Of course, it's not efficient to draw a large picture when only a portion of it is being displayed. The **Paint** member function can be optimized to draw only the part of the picture being displayed, as described at the end of this section.

A scrolling example

This example illustrates how to build a complete application that draws a scrollable graphics design. The example draws a series of rectangles that increase in size, so that the entire picture will not fit in the client area of a window drawn on a regular VGA screen. To view different parts of the design, you can use the scroll bars or you can auto-scroll the picture by holding the left mouse button down within the client area and then moving it out of the area.

This is SCROLAPP.CPP

```
#include <owl.h>

// Declare TScrollApp, a TApplication descendant
class TScrollApp : public TApplication {
public:
  TScrollApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
             LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
             hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

// Declare TScrollWindow, a TWindow descendant
class TScrollWindow : public TWindow
{
public:
  TScrollWindow(LPSTR ATitle);
  virtual void Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo);
};

/* Constructor for a TScrollWindow, sets scroll styles and
   constructs the Scroller object. */
TScrollWindow::TScrollWindow(LPSTR ATitle) : TWindow(NULL, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 60);
}

/* Responds to an incoming "paint" message by redrawing boxes. Note
   that the Scroller's BeginView method, which sets the viewport
   origin relative to the present scroll position, has already been
   called. */
void TScrollWindow::Paint(HDC PaintDC,PAINTSTRUCT&)
{
  int X1, Y1, I;

  for (I = 0; I <= 49; ++I)
  {
    X1 = 10 + I*8;
```

```
    Y1 = 30 + I*5;
    Rectangle(PaintDC, X1, Y1, X1 + X1, X1 + Y1 * 2);
  }
}

// Construct the TScrollApp's MainWindow of type TScrollWindow
void TScrollApp::InitMainWindow()
{
  MainWindow = new TScrollWindow("Boxes");
}

// Run the ScrollApp
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TScrollApp ScrollApp("ScrollApp", hInstance, hPrevInstance,
                       lpCmdLine, nCmdShow);
  ScrollApp.Run();
  return ScrollApp.Status;
}
```

Auto-scrolling and tracking

TScroller's auto-scrolling feature is enabled by default, but can be turned off by setting its *AutoMode* data member to FALSE. The owning window could do this in its constructor, after constructing the **TScroller** object:

```
Scroller = new TScroller(this, 8, 15, 80, 60);
Scroller->AutoMode = FALSE;
```

If this is done, scrolling can only take place using the scroll bars.

One useful feature of auto-scrolling is the fact that the farther you move the mouse out of the window's client area, the faster the window scrolls. Depending on how far out the mouse is, the window scrolls in increments as small as the line value and as large as the page value.

In addition to auto-scrolling, the example program described previously will "tracks" the scrolling requests made by moving the scrolling button, or "thumb." In other words, the picture moves as the thumb is moved. This feature gives instant feedback, so that the user can move to the desired part of the picture without having to raise the mouse button.

In some cases, however, tracking may be undesirable. For example, if you are displaying a large file of text, tracking can be slowed by repeatedly reading the disk and displaying text that

corresponds to each thumb movement. In this situation, it is better to disable tracking:

```
Scroller->TrackMode = FALSE;
```

That way, when the thumb is being moved, no scrolling takes place until the mouse button is released. At that point, the client area is scrolled just once to show the correct portion of the picture.

Modifying the scrolling units and range

In the examples so far, we assumed that the unit and range values were known at the time of **TScroller** construction. In many cases, however, this information is not known or can change, such as when the size of the information to be displayed changes. It may be necessary to set or change the range values (and perhaps the units) at a later time. If you don't know these values at construction, you can pass zero to the **TScroller** constructor.

The **SetRange** member function takes two **long** arguments, the number of horizontal and vertical units that determine the total scrolling range. **SetRange** should be used whenever the size of the picture changes. For example, when getting ready to display a picture 100 units wide and 300 units high, this command properly sets the scroll range:

```
Scroller->SetRange(100, 300);
```

If the units are not known when the **TScroller** object is initialized, their values must be set before scrolling can take place. For example, they could be set in the window's **SetupWindow** member function:

```
void ScrollWindow::SetupWindow()
{
  Twindow::SetupWindow();
  Scroller->XUnit = 10;
  Scroller->YUnit = 20;
}
```

Modifying the scrolling position

You can force scrolling by calling **TScroller**'s **ScrollTo** and **ScrollBy** member functions. Each of these takes two **long** arguments in terms of horizontal and vertical scrolling units. For

example, if it's necessary to reset the scroll position to the upper left corner of the picture, use **ScrollTo**:

```
Scroller->ScrollTo(0, 0);
```

As another example, if the picture is 400 units long in the vertical direction, the scroll position can be set to the middle of the picture in this way:

```
Scroller->ScrollTo(0, 200);
```

The **ScrollBy** member function moves the scroll position by a number of units up, down, left, or right. Negative values move the scroll position toward the origin, or the top left corner, and positive values move right and down. For example, this command scrolls right 10 units and down 20 units:

```
Scroller->ScrollBy(10, 20);
```

Setting the page size

By default, the page size (*XPage* and *YPage*) is set according to the size of the window's client area. The scroller is notified when its owning window's size changes. The owning window's **WMSize** member function calls its scroller's **SetPageSize** member function, which sets its data members *XPage* and *YPage* based on the current size of the window's client area and the values of *XUnit* and *YUnit*.

To redefine this mechanism, and set the page size directly, you must redefine your window object's inherited **WMSize** member function:

```
void TTestWindow::WMSize(RTMessage Msg)
{
  TWindow::WMSize(Msg);
}
```

Then you can set *XPage* and *YPage* directly from the window's constructor (or in the constructor of a class derived from **TScroller**):

```
TScrollWindow::TScrollWindow(PTWindowsObject AParent, LPSTR ATitle) :
                             TWindow(AParent, ATitle)
{
  Attr.Style |= WS_VSCROLL | WS_HSCROLL;
  Scroller = new TScroller(this, 8, 15, 80, 400);
  Scroller -> XPage = 40;
```

```
  Scroller -> YPage = 100;
}
```

Optimizing Paint member functions for scrolling

The previous example application draws 50 rectangles but does not attempt to determine if all of the rectangles are actually visible in the window's client area. This might result in wasted effort spent drawing non-visible graphics. The **TScroller::IsVisibleRect** function can be used in a window's **Paint** member function to optimize window painting.

The **ScrollWindow::Paint** member function described next uses **IsVisibleRect** to determine whether to call the Windows function **Rectangle**. **Rectangle** takes arguments in device units, while **IsVisibleRect** is in terms of scrolling units. For this reason, the values *X1* and *Y1*, the rectangle's origin, and (*X2-X1*) and (*Y2-Y1*), the rectangle's width and height, must be divided by the respective unit values before calling **IsVisibleRect**:

```
void TScrollWindow::Paint(HDC PaintDC, PAINTSTRUCT& PaintInfo)
{
  int X1, Y1, X2, Y2, I;

  for (I = 0; I <= 49; ++I)
  {
    X1 = 10 + I*8;
    Y1 = 30 + I*5;
    X2 = X1 + X1;
    Y2 = X1 + Y1 * 2;
    if (Scroller->IsVisibleRect(X1/Scroller->XUnit,
                                Y1/Scroller->YUnit,
                                (X2 - X1)/Scroller->XUnit,
                                (Y2 - Y1)/Scroller->YUnit))
      Rectangle(PaintDC, X1, Y1, X2, Y2);
  }
}
```

Edit windows and file windows

See "Building an ObjectWindows application" on page 20.

ObjectWindows provides two descendants of **TWindow** that are specialized windows for text editing. Instances of the **TEditWindow** class are text-editing windows that cannot read or write to a file. Instances of the **TFileWindow** class, derived from **TEditWindow**, define the additional behavior necessary to perform file I/O. You can use instances of these classes in your own

applications. Or, you can create specialized text editors that are derived from these classes. Programs using edit windows or file windows must include editwnd.h or filewnd.h respectively and must include certain ObjectWindows-supplied resources.

Edit windows

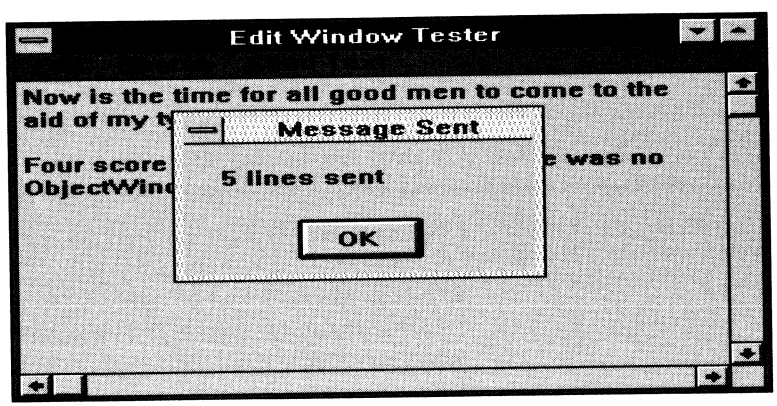
Text in a **TEditWindow** is actually contained and processed in a child **TEdit** control. It is the **TEdit** control that does most of the work, having been supplied by ObjectWindows with extensive text-editing capabilities.

TEditWindow's constructor sets its **Editor** data member to reference the instance of **TEdit** that it constructs. Its **WMSize** member function, invoked whenever the **TEditWindow** is sized, sizes its **Editor** so that it fills the client area of the **TEditWindow**. Its **WMSetFocus** member passes the input focus to **Editor** whenever the **TEditWindow** receives the focus.

TEditWindow does pitch in to add additional functionality to the basic **Search** behavior built in to a **TEdit**. It is responsible for the bulk of the Search and Replace functionality that is provided to the user of a **TEditWindow**. While TEdit controls know how to Search the text they contain, it's the **TEditWindow** that provides the ability to Replace text. **TEditWindow**s also provide "next" functionality, through maintenance of a **TSearchStruct**. **TEditWindow**s are also solely responsible for the Search and Replace user interface.

The following program, EWNDTEST.CPP, uses an edit window to let a user edit text for a simple (nonfunctional) electronic mail application. Figure 10.3 shows this application.

Figure 10.3
An edit window



This is EWNDTEST.CPP

```
#include <owl.h>
#include <stdio.h>
#include <editwnd.h>
#include "ewndtest.h"

class TTestApp : public TApplication
{
public:
  TTestApp(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance, LPSTR
           lpCmdLine, int nCmdShow) : TApplication(AName, hInstance,
           hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
  virtual void InitInstance();
};

class TTestWindow : public TEditWindow
{
public:
  TTestWindow(PTWindowsObject AParent, LPSTR ATitle);
  virtual void CMSendText(RTMessage Msg)
    = [CM_FIRST + CM_SENDTEXT];
};

TTestWindow::TTestWindow(PTWindowsObject AParent, LPSTR ATitle) :
                         TEditWindow(AParent, ATitle)
{
  AssignMenu("COMMANDS");
}

void TTestWindow::CMSendText(RTMessage)
{
  int Lines;
  char Text[20];

  Lines = Editor->GetNumLines();
  sprintf(Text, "%d lines sent", Lines);
  MessageBox(HWindow, Text, "Message Sent", MB_OK);
}

void TTestApp::InitMainWindow()
{
  MainWindow = new TTestWindow(NULL, Name);
}

void TTestApp::InitInstance()
{
  TApplication::InitInstance();
  HAccTable = LoadAccelerators(hInstance, "EDITCOMMANDS");
}
```

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR
                   lpCmdLine, int nCmdShow)
{
  TTestApp TestApp("Edit Window Tester", hInstance, hPrevInstance,
                   lpCmdLine, nCmdShow);
  TestApp.Run();
  return TestApp.Status;
}
```

This is EWNDTEST.RC

```
#include "windows.h"
#include "ewndtest.h"
#include "owlrc.h"

rcinclude stdwnds.dlg
rcinclude editacc.rc

COMMANDS MENU LOADONCALL MOVEABLE PURE DISCARDABLE
BEGIN
  POPUP "&Edit"
  BEGIN
    MenuItem  "&Undo\aAlt+BkSp", CM_EDITUNDO
    MenuItem  SEPARATOR
    MenuItem  "&Cut\aShift+Del", CM_EDITCUT
    MenuItem  "C&opy\aIns", CM_EDITCOPY
    MenuItem  "&Paste\aShift+Ins", CM_EDITPASTE
    MenuItem  "&Delete\aDel", CM_EDITDELETE
    MenuItem  "C&lear All\aCtrl+Del", CM_EDITCLEAR
  END
  POPUP "&Search"
  BEGIN
    MenuItem  "&Find...", CM_EDITFIND
    MenuItem  "&Replace...", CM_EDITREPLACE
    MenuItem  "&Next\aF3", CM_EDITFINDNEXT
  END
  MenuItem  "Send &Text", CM_SENDTEXT
END
```

File windows

TFileWindow's constructor takes the name of a file as an argument. **TFileWindow::SetupWindow** sets up the file window by reading the contents of this file into its **Editor**, the **TEdit** control it inherits from **TEditWindow**. **TFileWindow** also redefines **CanClose** to prompt the user to save current edits, if any, before the file window closes.

TFileWindow has four member functions that respond to requests from the user to open and save files: **CMFileNew**, **CMFileOpen**,

CMFileSave, and **CMFileSaveAs**. The user issues these requests by selecting corresponding menu items with the identifiers listed in Table 10.4.

Table 10.4
File window member functions and menu IDs

Member function	Menu ID to invoke
CMFileNew	CM_FILENEW
CMFileOpen	CM_FILEOPEN
CMFileSave	CM_FILESAVE
CMFileSaveAs	CM_FILESAVEAS

Any of these requests that need to get a file name from the user use an instance of a **TFileDialog** to let the user specify the drive, directory, and file name.

You can use file windows as simple standalone text editors without modification. However, sometimes you will want to derive your own classes from **TFileWindow** to provide additional functionality. For example, a class you derive from **TFileWindow** may provide the ability to change the font of the text that is displayed in the *Editor*. Normally, you would present this option to the user as a drop-down submenu of a menu that you create. To use the user interface for file I/O that is provided by **TFileWindow**, create menu items with the identifiers listed in Table 10.4.

C H A P T E R

11

Dialog objects

A dialog is an interface element whose creation attributes are specified in a Windows resource file. The creation attributes in the dialog's resource definition define the appearance and placement of the dialog and its controls.

Dialogs are commonly used as child windows to perform a specific input-related task. For example, a child dialog could be used to accept user input to configure a printer. ObjectWindows supplies three classes derived from **TDialog**. These classes perform common tasks: **TInputDialog** retrieves a single line of text from the user, **TFileDialog** retrieves a file name, and **TSearchDialog** retrieves search and replace text for an edit window.

Using dialog resources

A dialog interface element must have a resource definition in a Windows resource file. The resource definition, which has a unique identifier, specifies the appearance and placement of the dialog element and its controls. The resource definition does not specify the behavior of the dialog.

See page 20 for a discussion of building an ObjectWindows application with resources. Be sure to append your resources to your application.

In an ObjectWindows application, a **TDialog** object defines the behavior of a dialog. Its constructor takes a resource definition identifier as a parameter. This resource definition is used when an interface element is created to be associated with the **TDialog**.

Using a child dialog object

Managing a child dialog object is a lot like managing a pop-up window. Dialogs and pop-up windows are often created, displayed, and destroyed many times during the life of their parent windows. Dialogs and pop-up windows that appear for only a short time can be constructed, created, and deleted in one member function without being stored as data members of their parent. Those that are created for a longer period of time are normally stored as data members of their parents.

Constructing and initializing child dialog objects

The constructor of a **TDialog** requires two parameters: a pointer to a parent window object, and the dialog's resource identifier. **TDialog** overloads its constructors, to accept either an **LPSTR** or **int** resource identifier:

```
ADialog = new TSampleDialog(this, "SampleDialog");
```

or

```
ADialog = new TSampleDialog(this, 101);
```

It's generally good practice to use constants for your identifiers:

```
#define ID_SAMPLERESOURCE 101
⋮
ADialog = new TSampleDialog(this, ID_SAMPLERESOURCE);
```

Dialog objects should always be constructed on the heap (not on the stack). ObjectWindows will automatically delete the dialog object when the corresponding Windows element is closed.

Creating and executing dialogs

The association between a child dialog object and an interface element is made when the **TDialog** is *executed* or *created*.

A child dialog is *executed* when further processing depends on input from the user; processing halts until input is retrieved and the dialog is closed. For example, the closing of a text-editing window would execute a dialog to determine whether unsaved edits should be saved. Further processing (saving or not saving

the edits) would depend on user input. Processing would halt (the window would not be closed) until the input was retrieved. In fact, while a dialog is executing, its parent window is disabled and is unable to respond to user input.

Alternatively, a child dialog is *created* when user input can affect further processing, but processing can continue without it. For example, a text-editing window might create a search dialog to retrieve a search string the first time the user requests a search. If not explicitly closed by the user, the same dialog could be used to retrieve new search strings for subsequent search requests, but normal processing (text-editing) would still continue. A created dialog does not disable its parent window.

Executing a dialog differs from creating a dialog in an additional respect: When you execute a dialog, the dialog returns a value signaling an option chosen by the user. Simple dialogs often display questioning prompts, and have OK and Cancel buttons. When the user clicks either button, the dialog terminates and returns the identifier of the button clicked as the result of its execution.

See the section titled "Transferring control data" in Chapter 12.

Windows, however, allows only an integer value to be returned by an executed dialog. To retrieve more complex input, you can customize the data transfer mechanism supplied by ObjectWindows or supply and fill a transfer buffer of your own definition in your derived dialog class.

Modal and modeless dialogs

A dialog that is executed is said to be a *modal* dialog; a *modeless* dialog is a dialog that is created. A **TDialog** can be associated with either a modal or a modeless dialog through calls to its **Execute** and **Create** member functions, respectively. However, these member functions should not be called directly. Invoke the **ExecDialog** member function of the application object to safely *execute* a modal dialog; invoke its **MakeWindow** method to safely *create* a modeless dialog. **ExecDialog** and **MakeWindow** are inherited from **TModule**.

```
// Execute a modal dialog
if (GetModule()->ExecDialog(new TSampleDialog(this,
                            "SampleDialog")) == IDOK)
  // do something

// Create a modeless dialog
GetModule()->MakeWindow(new TSampleDialog(this, "SampleDialog"));
```

Closing a child dialog

See "Transferring control data" on page 187.

Close your modal or modeless dialog by calling its **CloseWindow** member function (which the **Ok** message response member function does). **CloseWindow** closes the dialog if **CanClose** reports it's OK to do so. **CloseWindow** for a modal dialog, by default, transfers the dialog's data. If you want to unconditionally close a dialog (without getting permission from **CanClose**), call **ShutDownWindow** (which the **Cancel** message response member function does).

See "Transferring the data" on page 190.

You won't normally redefine **ShutDownWindow**, but you might want to redefine **CloseWindow** to have a modeless dialog call **TransferData**. You might also want to redefine **CanClose** to validate the data before closing. For example, if the text in an edit control is invalid, display a message box.

For a modal dialog, the value passed to **CloseWindow** or **ShutDownWindow** becomes the return value of **Execute** and is also returned by **TModule::ExecDialog**. If no value is specified, the default return value is IDCANCEL. For a modeless dialog, you don't need to pass a value, since a modeless dialog has no return value.

Using a dialog as a main window

You can use a modeless dialog as the main window of your application. You may want to do this if your main window contains a large number of controls. By using a dialog as your main window, you'll be able to use a dialog resource editor to more easily define the creation attributes of your window and its controls. Use a class derived from **TDialog** as a main window the same way as you would for a class derived from **TWindow**, constructing it in **InitMainWindow**. The CALC.CPP program in the EXAMPLES directory uses a modeless calculator dialog as its main window.

You can also use a modeless dialog as the main window of your application if you want to separate the interface element specification from your application. That way, you can modify the appearance of your main window without changing or having to recompile your application.

For example, a calculator program might have a modeless dialog as a main window, where the calculator's buttons are specified as button controls in a dialog resource. This would allow the main window to also have a menu, icon, and cursor.

You will encounter some difficulty if you want to respond to accelerators in an application that has a dialog as a main window. Windows doesn't fully support the use of accelerators in modeless dialogs. However, there are tricks you can use to get around this limitation. The CALC.CPP program uses one approach.

Defining a Windows class for your modeless dialog

You've already encountered the concept of *registration attributes* and the process of defining these attributes for a window in the section titled "Windows class registration" in Chapter 10, "Window objects." In summary, Windows requires that certain attributes of a window be registered before it is created. These registration attributes include an icon and a cursor. ObjectWindows registers default attributes for your windows; to change these defaults, you must redefine **GetWindowClass** and **GetClassName**.

To modify the registration attributes for your dialogs, redefine **GetWindowClass** and **GetClassName**, just as you would for a window. Additionally, you'll need to specify the name of the Windows class of your dialog in the resource definition of your dialog. This is the name which your dialog's **GetClassName** member function returns.

If you do not specify a Windows class name in your dialog's resource definition, your dialog will have the default registration attributes defined by Windows. If you do specify a Windows class in your dialog's resource definition, you will not be able to execute the dialog modally.

Control manipulation and message processing

Windows provides little support for the creation and management of controls by a parent window; window and control objects handle these tasks in an ObjectWindows application. (Chapter 12, "Control objects," provides details on the use of controls in a window.)

Windows does handle the creation of controls in a dialog, as well as some support for their management. Therefore, in ObjectWindows the controls of a dialog are not, by default, associated with control objects. However, you may want to associate the

controls of your dialogs with ObjectWindows control objects to facilitate dialog/control communication.

A dialog object and its control elements carry on a two-way communication. In one direction, the dialog needs to manipulate and query its controls. For example, a dialog might need to fill a list box control, or retrieve text from an edit control. In the other direction, the controls of a dialog need to notify the dialog when an event has occurred that the dialog may need to be aware of. For example, a button control notifies its parent dialog when it is pressed.

Manipulating dialog controls

Windows provides a set of control messages, and default responses, for each type of control. For example, the default response for a list box to an LB_GETCURSEL message is to return the index of the selected item. A dialog communicates with its controls by sending control messages specifying a control identifier, a message identifier, and message-specific parameters.

In ObjectWindows, you send a message to a control of a dialog by calling the dialog's **SendDlgItemMsg** member function, which expects four parameters:

- the identifier of the control
- the identifier of the message
- a **WORD** parameter containing the specifics of the message (related data)
- a **LONG** parameter containing more specifics

In the following sample code, **TestDialog**'s **SendDlgItemMsg** member function is called to add "Item 1" to its list box control (by sending an LB_ADDSTRING message).

```
void TTestDialog::FillListBox()
{
  SendDlgItemMsg(ID_LISTBOX, LB_ADDSTRING, NULL, (DWORD) "Item 1");
}
```

You can facilitate communication between your dialog object and its controls by associating the controls with control objects that are members of the dialog class. (Remember that the controls of a dialog are not, by default, associated with ObjectWindows control objects.) After doing so, you'll be able to manipulate the control using the object's data and function members, OOP-style, instead of having to send messages to the control's interface element through the Windows API.

To do so, construct a control object in the constructor of your dialog. Use the constructor of the control object which accepts only the parent object and the resource identifier as parameters. (There's no need to pass creation attributes since Windows has already created the element.) Now you'll be able to manipulate and query the dialog's controls just as you would the controls of a window. Related changes to the declaration, to the constructor, and to the **FillListBox** member function of **TTestDialog** are shown next.

```
class TTestDialog : public TDialog {
public:
:
  PTListBox ListBox;
  TTestDialog(PTWindowsObject AParent, LPSTR AName);
:
};

TTestDialog::TTestDialog(PTWindowsObject AParent, LPSTR AName) :
                         TDialog(AParent, AName)
{
  ListBox = new TListBox(this, ID_LISTBOX);
}

void TTestDialog::FillListBox()
{
  ListBox->AddString("Item 1");
}
```

Responding to control notification messages

A control sends a control notification message to its parent when an event occurs that the parent may need to be aware of. In ObjectWindows, a dialog object specifies a response to messages sent by a particular control by defining a child-ID-based message response member function. This method is automatically invoked when the dialog receives a notification message from the identified child window.

In the following example declaration, **TTestDialog** declares child-ID-based response methods for two of its controls:

```
class TTestDialog : public TDialog {
:
void HandleButtonMsg(RTMessage Msg) =
  [ID_FIRST + ID_BUTTON];
void HandleListBoxMsg(RTMessage Msg) =
  [ID_FIRST + ID_LISTBOX];
};
```

Each control notification message comes with a notification code, an integer constant, which identifies the action that occurred. For example, when an item in a list box is selected, a control notification message is sent with an LBN_SELCHANGE notification code; a BN_CLICKED code is passed when a button is clicked. (The notification code is passed in *Msg.LP.Hi*, the high-order word of *Msg.LParam*).

Usually, the responses specified in your child-ID-based message response member function depend on the notification code passed in *Msg.LP.Hi*, as shown in the following example:

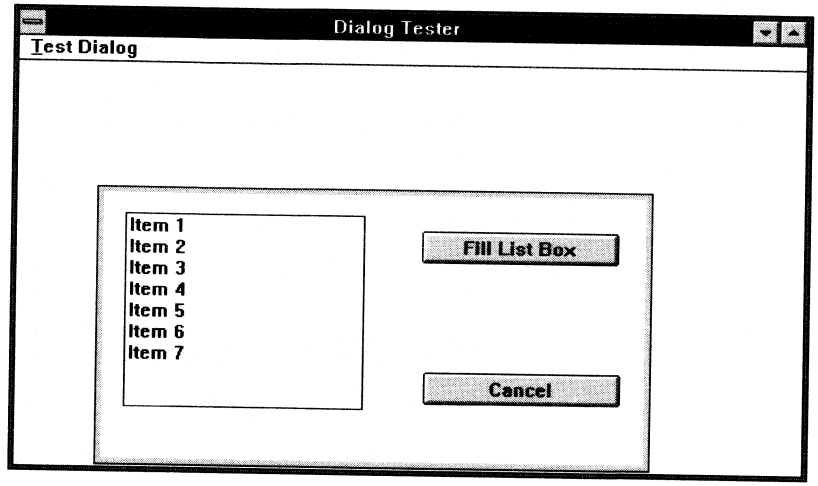
```
void TTestDialog::HandleListBoxMsg(RTMessage Msg) {
  switch (Msg.LP.Hi)
  {
    case LBN_SELCHANGE :
      // Handle selection change
      break;
    case LBN_DBLCLK :
      // Handle selection double-click
      break;
:
  }
}
```

You've seen how to define responses to control notification messages for your dialog objects. However, you may prefer to have your dialog's controls respond to their own notification messages. For example, you might want a button in your dialog to perform some action when pressed. To do so, you'll need to define a notify-based member function in a control class that you define. Then, you associate an ObjectWindows control object with the control of your dialog (described in the preceding section).

Example of dialog/control communication

In the following program, a **TTestDialog** class is derived from **TDialog**. **TTestDialog** defines **HandleButtonMsg** and **HandleListBoxMsg**, which respond to notification messages sent to the dialog by its button control and its list box control. **HandleButtonMsg**, which is automatically invoked when the button is pressed, responds by filling the list box control with text items through calls to the dialog's **SendDlgItemMsg** member function. The **TTestDialog** is executed as a modal dialog in the **ExecDialog** member function of its parent window.

Figure 11.1
A dialog application



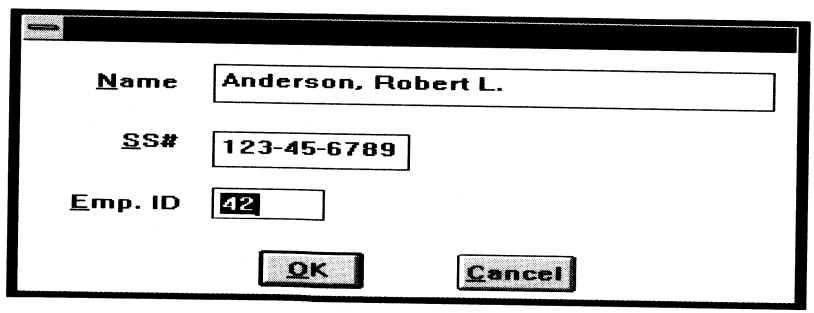
The complete program, DIALTEST.CPP, can be found in the EXAMPLES subdirectory.

Extended example of using dialogs

VDLGAPP.CPP defines a complex dialog that asks the user for text input and then validates the input. It displays a dialog box with edit fields for an employee's name, social security number, and employee ID.

When the user clicks the OK button, the **CanClose** member function verifies that a valid name, social security number, and employee ID have been entered before allowing the dialog to close. Figure 11.2 shows this dialog.

Figure 11.2
A dialog that validates user input



The complete listing of VDLGAPP.CPP can be found in the EXAMPLES subdirectory.

Input dialogs

Input dialogs are stock dialogs provided with ObjectWindows. They display a prompt and retrieve a single line of input text from a user. This is a common task in a windowing application; you'll therefore probably find many uses for input dialogs in your own applications.

To use input dialogs, you must include inputdia.h in your program and INPUTDIA.DLG in your program's .RC file.

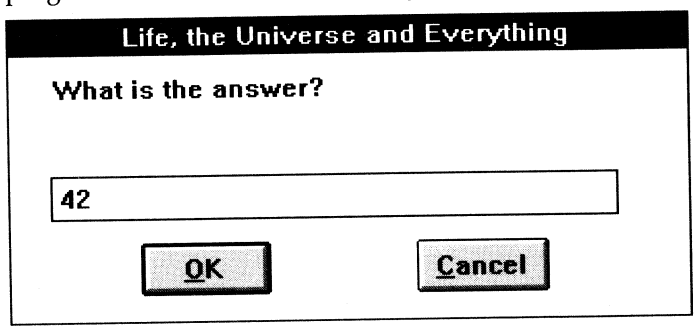


Figure 11.3
An input dialog

The constructor of an input dialog accepts five parameters:

- a pointer to a parent window object
- a pointer to text displayed as the caption of the dialog
- a pointer to text displayed as the prompt
- a pointer to a text buffer to be filled with the text input by the user
- the size of the text input buffer

An input dialog is constructed in the following example. When the OK button of the input dialog is pressed, *EditText* is filled with the text the user has entered into the edit control of the dialog.

```
PTInputDialog AnInputDialog;
char EditText[79];

strcpy(EditText, "");
AnInputDialog = new TInputDialog(this, "Caption", "Prompt", EditText,
                                 sizeof(EditText));
```

The text buffer passed to the constructor of the input dialog is expected to hold a default text string to be displayed when the input dialog is initially displayed. Be sure to set the contents of the text buffer to some string (including a null string).

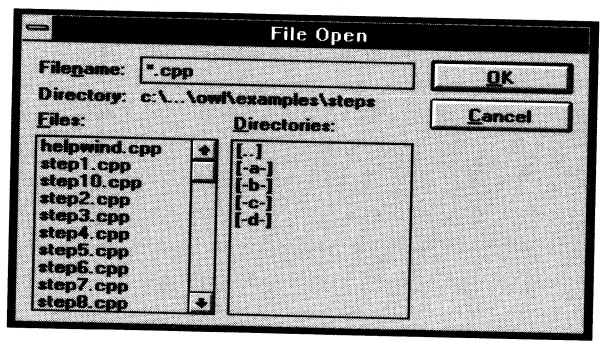
The pointers to the text items of the input dialog must remain valid for the life of the dialog. This is particularly important to remember when running an input dialog as a modeless dialog. Most often, however, you'll run your input dialogs as modal dialogs, as demonstrated in the following example:

```
void TSampleWindow::GetName()
{
  char EditText[79];

  strcpy(EditText, "Frank Borland");
  if (ExecDialog(new TInputDialog(this, "Data Entry", "Enter Name",
      EditText, sizeof(EditText))) == IDOK)
    MessageBox(HWindow, EditText, "Name is:", MB_OK);
  else MessageBox(HWindow, EditText, "Name is still:", MB_OK);
}
```

File dialogs

File dialogs are another type of stock dialog provided with ObjectWindows in the **TFileDialog** class. Use a file dialog every time you want to prompt the user for a file name, such as in the File Open, Save, and Save As functions many applications feature. See Figure 11.4.

Figure 11.4
A file dialog



To use **TFileDialog**, you must include filedial.h in your program and FILEDIAL.DLG in your program's .RC file.

The constructor of a file dialog box takes three parameters: a pointer to a parent window, a resource identifier, and a file name or mask (depending on whether the file dialog is for opening or saving a file). The resource identifier passed is the identifier of either the standard Open or the standard Save As file dialog box (SD_FILEOPEN or SD_FILESAVE). A default file mask is passed

in the file-name parameter for an Open file dialog; it contains a default file name for a Save As file dialog. The file name parameter is also used as a return buffer for the file name selected by the user for both types of dialogs.

An Open file dialog box is executed in the following example code. "*.*" is passed as the file mask to the constructor of the file dialog, so that all files in the current directory will be listed in the file name list box when the file dialog is initially displayed:

```
char TempName[MAXPATH];
⋮
_fstrcpy(TempName, "*.*");
if (GetApplication()->ExecDialog(new TFileDialog(this, SD_FILEOPEN,
                                 TempName)) == IDOK))
// open the file named in TempName
⋮
```

A Save As file dialog box is executed in the example code next. A null string is passed as the default file name to the constructor of the file dialog, so that no default named file will be selected when the file dialog is initially displayed:

```
char TempName[MAXPATH];
⋮
TempName[0] = '\0';
if (GetApplication()->ExecDialog(new TFileDialog(this, SD_FILESAVE,
                                 TempName)) == IDOK)
// save to the file named in TempName
⋮
```

C H A P T E R

12

Control objects

User interface elements that facilitate the transfer of user input are collectively referred to as *control elements*. *Check boxes* and *list boxes*, familiar to users of Windows applications, are two types of control elements. To Windows, controls are simply specialized windows; in ObjectWindows, therefore, **TControl** is derived from **TWindow**.

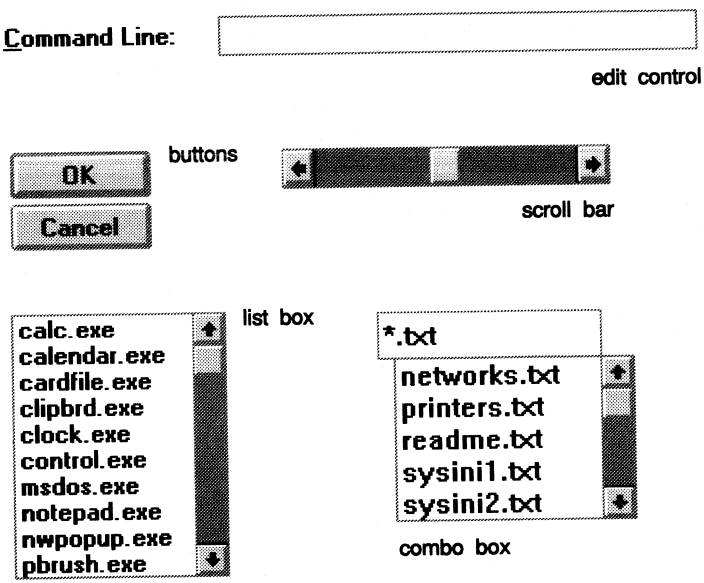
Classes derived from **TControl** are representatives of the Windows control elements. In your applications, you'll use instances of its derived classes, described in the following table:

Table 12.1: Windows controls supported by ObjectWindows

Control	Class	Use
List box	**TListBox**	A scrollable list of items, such as files, from which to choose.
Scroll bar	**TScrollBar**	A standalone scroll bar similar in appearance to those in scrolling windows and list boxes.
Push button	**TButton**	A push button, with associated text.
Check box	**TCheckBox**	A box that can be checked or unchecked, with associated text.
Radio button	**TRadioButton**	A button that can be selected or not. Usually used in mutually exclusive groups.
Group box	**TGroupBox**	A static rectangle with text in the upper left corner used to group other controls.
Edit control	**TEdit**	A field for the user to input text.
Static control	**TStatic**	A text field that cannot be modified by the user.
Combo box	**TComboBox**	A combination of a list box and edit control.

Figure 12.1 shows the screen appearance of some typical controls.

Figure 12.1
Some sample controls



For all the control types listed previously, ObjectWindows defines classes to be used as child windows inside other windows. Controls that are a part of a dialog box are control interface elements specified in a dialog resource and need no corresponding ObjectWindows object (see Chapter 11, "Dialog objects"). This chapter outlines the functionality of the supplied control objects and shows how they might be used inside of a window.

Use of control objects

The life of a control usually parallels the life of its parent window; controls are normally created and destroyed along with their parent. This behavior is supplied automatically in ObjectWindows. Each child window in a parent window's child list is created in the process of creating the parent and destroyed when the parent is destroyed.

Constructing and creating controls

You'll almost always construct your control objects in the constructor of their parent window, but it is not necessary to maintain references to the children you construct as data members of the parent. A child window reference can always be obtained by calling its parent window's **ChildWithID** method (assuming you've given each of the children a unique identifier in the *AnID* parameter to a constructor of all **TControl**-derived classes). If you have a child window that your code often accesses, you may find it more convenient to maintain a pointer as a data member of its parent.

The constructors of control objects usually accept at least six parameters: a pointer to the parent window object, the control's ID, the x- and y-coordinates of its upper left corner (relative to the parent's client area), and its width and height. These parameters are used to set creation attributes of the control object, which are maintained in its **Attr** structure. The styles that the **TControl** constructor sets in *Attr.Style* include WS_CHILD, WS_VISIBLE, WS_TABSTOP, and WS_GROUP. This default can be changed in descendant control classes, as it is for some of the ObjectWindows control classes. See Table 10.1 for window attribute data members.

You should not directly call **Create** (or **GetApplication()->MakeWindow**) for each of your controls. They will be automatically created when the **SetupWindow** member function of **TWindowsObject** is invoked for the parent window. You can redefine **SetupWindow** in the derived class of your parent window to perform setup processing for its controls. If you do redefine **SetupWindow**, be sure to call the **SetupWindow** member function of the base class to create the controls.

To review, you optionally define data members for your parent window to hold pointers to its controls. Then, you construct the control object in the constructor of the parent window. Your last step is to redefine **SetupWindow** to perform required setup processing for the controls, if any. ObjectWindows will handle the rest.

Destroying and deleting controls

In ObjectWindows, the destruction and deletion of control windows is also the responsibility of the parent window. A

control element is automatically destroyed and its associated control object is automatically deleted when the parent window is destroyed and deleted.

Controls and message processing

Communication between a window object and its control objects is similar in some ways to the communication between a dialog object and its control elements. In both cases, the parent manipulates and queries its controls, and the controls send notification messages of control events to the parent. The parent can define child-ID-based response methods for these notification messages. However, the controls of a dialog are not, by default, associated with ObjectWindows control objects; control objects are always associated with the controls of a window. This section describes window/control communication. See Chapter 11, "Dialog objects," for a description of dialog/control communication.

Manipulating a window's controls

Windows manipulate and query their controls by calling control member functions.

When a window's control objects have corresponding data members in the parent window's object, it is simple to call control member functions:

```
ListBox->AddString("Scotts Valley");
```

When a control object pointer is not maintained as a data member, it must be retrieved from the parent's child list by calling the parent's **ChildWithID** member function. The following code fragment is from a member function of the parent window object.

```
PTListBox TheListBox;
⋮
TheListBox = (PListBox)ChildWithID(ID_LISTBOX);
TheListBox->AddString("Scotts Valley");
```

Responding to control notification messages

An event affecting a control of a window results in the control sending a *control notification message* to the window (see "Responding to control events" in Chapter 6). In most cases, the parent window handles notification messages from a control in child-ID-based response methods, as shown in the following example:

```
class TSampleWindow : public TWindow {
public:
  PTListBox ListBox;
  PTButton Button1, Button2;

  virtual void HandleListBoxMsg(RTMessage Msg) =
    [ID_FIRST + ID_LISTBOX];
  virtual void HandleButton1Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON1];
  virtual void HandleButton2Msg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON2];
}
```

In your response member functions, *Msg.LP.Hi* carries the control notification code, such as LBN_SELCHANGE and BN_CLICKED. Write a response to handle the important notification codes:

```
void TSampleWindow::HandleListBoxMsg(RTMessage Msg)
{
  switch (Msg.LP.Hi) {
    case LBN_SELCHANGE:
      // Handle selection change
      break;

    case LBN_DBLCLK:
      // Handle selection double-click
      break:
  }
}
```

There are times, however, when you want a control object itself to respond to a control notification message, thereby building a response behavior into the control. For example, you might want to design a button that changes its text when the user clicks it, or an edit control that allows only digits to be entered. In these special cases, you want the control class to supply the desired behavior, rather than duplicating it in each parent window. The result is a standalone class that can be reused many times in many applications.

To program a control to respond to its own notification messages, define a *notify-based* response member function in a **TControl** derived class, as shown here:

```
class TSpecializedListBox : public TListBox {
public:
  virtual void LBNSelChange(RTMessage Msg) =
    [NF_FIRST + LBN_SELCHANGE];
  virtual void LBNDblClick(RTMessage Msg) =
```

```
    [NF_FIRST + LBN_DBLCLICK];
}
```

Be sure to use the sum of NF_FIRST and the control notification code in the function declaration extension for your notify-based response member functions.

The definition of **TSpecializedListBox::LBNSelChange** follows:

```
void TSpecializedListBox::LBNSelChange(RTMessage Msg)
{
  // Handle selection change
}
```

If you would like both the control and the parent window to respond to the notification message, call the control's **DefNotificationProc** method from its notify-based response member functions.

Control focus and the keyboard

Have you ever used the keyboard to shift the input focus from control to control in a dialog box? *Tab* can be used to cycle from control to control (in the order of creation). Additionally, the arrow keys can be used to shift the focus between the radio buttons within a group box, changing the selection in the process. In ObjectWindows, this keyboard interface can be emulated for the controls of your windows. To enable this "keyboard handling" functionality, call the **EnableKBHandler** member function of your window in its constructor.

List box controls

Using a list box is the simplest way to ask the user of a Windows program to pick something from a list. For example, you can ask the user to choose from a list of files, printers, shapes, or fruit, depending on the nature of the program. List boxes are encapsulated by the class **TListBox**. **TListBox** defines member functions for four purposes: creating list boxes, modifying the list of items, inquiring about the list of items, and finding out which item the user selected.

Constructing and creating list boxes

One of **TListBox**'s constructors takes a parent window, an ID, and the control's *X*, *Y*, *W*, and *H* dimensions as parameters. For example, the following statement might appear in a parent window constructor:

```
ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
```

TListBox's constructor invokes **TControl**'s constructor, then adds LBS_STANDARD to the default styles specified for the list box in *Attr.Style*. LBS_STANDARD is a combination of styles that specify that the list box will

- maintain its items in alphabetical order (LBS_SORT)
- notify its parent when list box events occur (LBS_NOTIFY)
- have a border (WS_BORDER)
- have a vertical scroll bar (WS_VSCROLL)

You can modify *Attr.Style*, which contains the default styles, after you've constructed your list box. For example, if you want 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;
```

You can also modify *Attr.Style* in the constructor of a **TListBox** derived class.

As with all child windows, list box control elements are automatically created by their parent windows.

Modifying list boxes

After you create a list box, you need to fill it with list items, which must be strings or owner-drawn. Later, you might want to add or remove items, or clear the list completely. To fill or add items, call the list box's **AddString** member function:

```
ListBox->AddString("Item 1");
ListBox->AddString("Item 2");
ListBox->AddString("Item 3");
ListBox->AddString("Item 4");
ListBox->AddString("Item 5");
ListBox->AddString("Item 6");
```

If **AddString** is unsuccessful, it returns a negative value.

A list box keeps a list of strings much like an array of strings; both keep their elements at an index. If *ListBox* were an array, "Item 1" would be stored at *ListBox[0]*, "Item 2" at *ListBox[1]*, "Item 3" at *ListBox[2]*, and so on. Every time you call **AddString**, the specified string is added in alphabetical order by default, or at the end of the list if the style excludes LBS_SORT.

Regardless of whether your list box is sorted, you can also insert a new element at a specified index with **InsertString**:

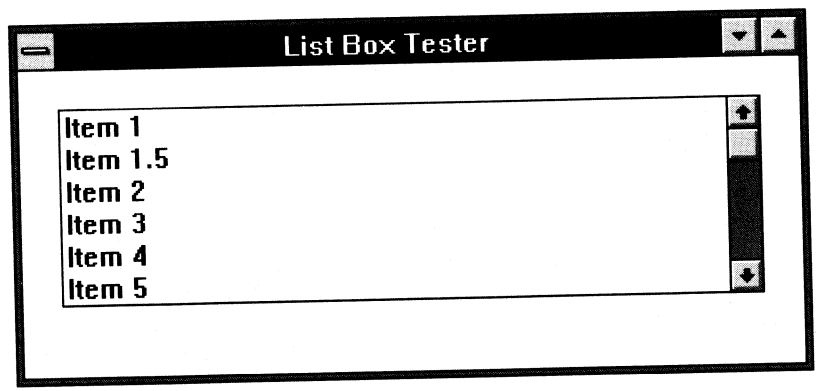
```
ListBox->InsertString("Item 1.5", 1);
```

This also moves the strings stored at each index greater or equal to 1 up one notch. The seven preceding member function calls result in the following list:

Index	Item
0	"Item 1"
1	"Item 1.5"
2	"Item 2"
3	"Item 3"
4	"Item 4"
5	"Item 5"
6	"Item 6"

See Figure 12.2.

Figure 12.2
A filled list box



Inserting a new string at index –1 appends the string to the end of the list.

To remove an element, call **DeleteString**. The following member function call deletes the string at index 1 ('Item 1.5') and moves the strings at an index higher than 1 down one index:

```
ListBox->DeleteString(1);
```

Finally, **ClearList** deletes every string in the list box:

```
ListBox->ClearList();
```

Querying list boxes

There are six member functions you can call to find out information about the list held by a list box object.

- **GetCount** returns the number of items in the list.
- **FindString** and **FindExactString** both search the list box for the specified string, starting at the index specified, wrapping to the beginning of the list if necessary. **FindString** searches for a string in the list box that *contains* the specified string while **FindExactString** searches for a string that *begins* with the specified string.
- **GetString** gets the string located in the list at the index specified by an integer argument. An **LPSTR** argument points to a buffer to receive the string. *GetString* also returns an integer representing the length of the string.
- **GetStringLen** simply returns the length of the string at the specified index, but doesn't return the string itself.
- **GetSelIndex** returns the index of the currently selected string, for single selection list boxes only.

Getting selections from a list box

The user can do three things with a list box: scroll through the list, single-click an item, and double-click an item. When a list-box user action takes place, Windows sends a *list box notification* message to the list box's parent window.

Every list box notification (*LBN*) message contains a list box notification code (an integer constant) in *Msg.LP.Hi* to specify the nature of the action. The following table summarizes the most common *LBN* codes:

Table 12.2
List box notification messages

WParam	Action
LBN_SELCHANGE	An option has been selected with a single mouse click.
LBN_DBLCLK	An option has been selected with a double mouse click.
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 and LBN_DBLCLK.

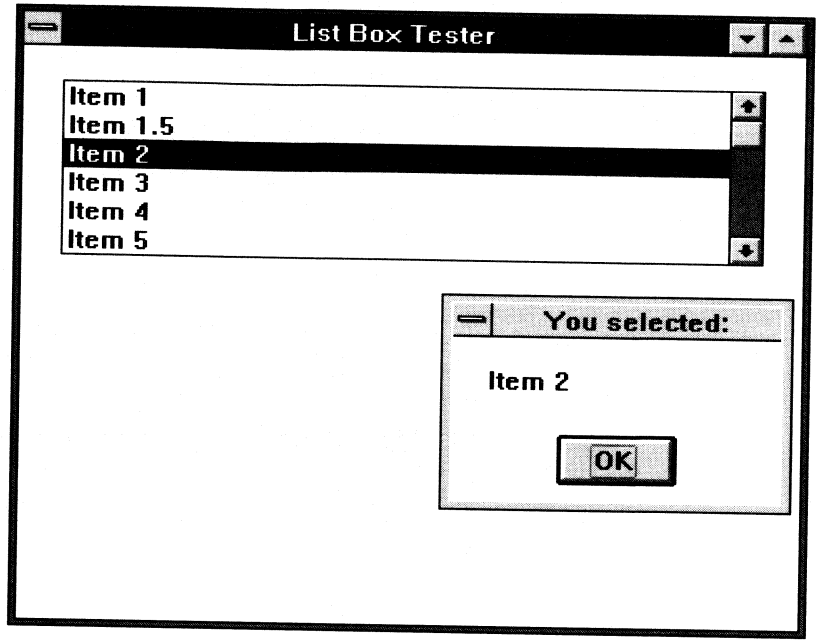
Your parent window usually handles notification messages from the list box in a child-ID-based response member function that you define. Here's a sample parent window response member function that handles notification messages from a list box:

This code fragment is from LBOXTEST.CPP.

```
void TSampleWindow::HandleListBoxMsg(RTMessage Msg)
{
  int Idx;
  char Str[10];

  if (Msg.LP.Hi == LBN_SELCHANGE)
  {
    Idx = ListBox->GetSelIndex();
    if (ListBox->GetStringLen(Idx) <= sizeof(Str))
    {
      ListBox->GetSelString(Str, sizeof(Str));
      MessageBox(HWindow, Str, "You selected:", MB_OK);
    }
  }
}
```

The user has made a selection if the high-order word of *LParam* contains the LBN_SELCHANGE notification code. If so, it gets the length of the selected string, verifies that it will fit in the *Str* buffer, returns the string, and shows it in a message box (see Figure 12.3).

Figure 12.3
Responding to the user selecting a list box item



Selected items are manipulated by four **TListBox** member functions: **GetSelString**, **GetSelIndex**, **SetSelString**, and **SetSelIndex**. The **Get...** member functions get the string and index of the selected string. The **Set...** member functions bypass the user and force the selection of a particular item by specifying the string or index. This causes the item to come into view, if it isn't already.

The program **LBoxTest** creates a window with a list box. When the application is started, a list box is displayed in the main window. When the user selects a list box item, a dialog appears with the list item displayed. The main window maintains a reference to the list box control in its *ListBox* data member to facilitate manipulation of the control.

The complete file, LBOXTEST.CPP, can be found in the EXAMPLES subdirectory.

Combo boxes

A combo box control is a combination of two other controls: a list box and an edit control. It's used to retrieve user input in one of

two forms: as a selection in the list box or as text in the edit control.

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 cases, can *drop down* on request.

Three varieties of combo boxes

There are three types of combo boxes: simple, drop down, and drop down list. Figure 12.4 shows the appearance of the three combo box types, as well as that of a list box.

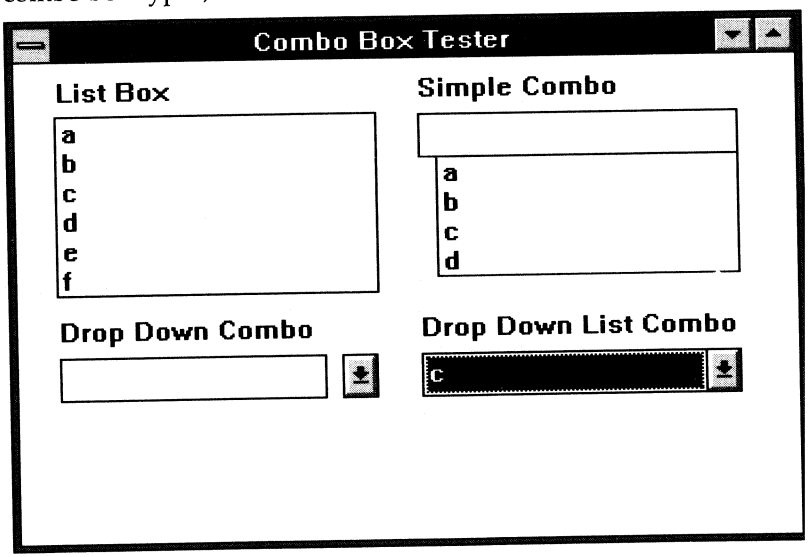


Figure 12.4
The three types of combo boxes and a list box

Simple combo boxes

All combo boxes display their edit area at all times. Some combo boxes, however, can show and hide their list box areas much like a turtle retracts its head into its shell. Simple combo boxes cannot hide their list box area; it is always displayed. A simple combo box contains an edit control, in which the user can enter and edit text. If the text entered matches one of the items in the list, the item is selected.

Drop down combo boxes

A drop down combo box behaves like a simple combo box with one exception. In its initial state, its list area is not displayed. It appears when the user clicks the down arrow to the right of the edit control. The list box is hidden when a selection is made. Drop down and drop down list combo boxes are useful when you are trying to fit a lot of controls into a small area. When they are not being used, they take up much less space than a simple combo box or a list box.

Drop down list combo boxes

The list box of a drop down list combo box behaves like the list box of a drop down combo box—it appears when needed and retracts when not needed. The two combo box types differ in the behavior of their edit areas. Drop down edit areas behave like regular edit controls, whereas drop down list edit areas display only the text from one of its list items, and cannot be modified directly by the user.

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 printers to which to print, you can only choose a printer accessible from your system (see Figure 12.5).

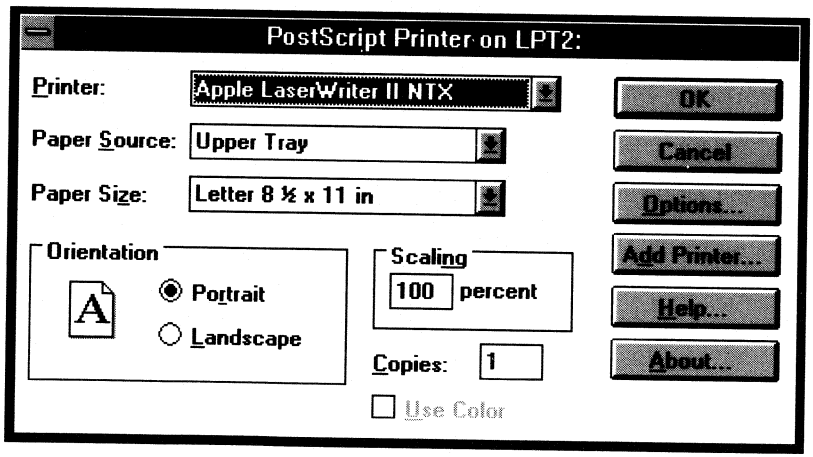


Figure 12.5
A drop down list combo box

On the other hand, drop down combo boxes can accept entries other than those found in the list. One 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's constructor accepts the usual control constructor parameters: a pointer to a parent window, a control ID, and location and size data. In addition, the **TComboBox** constructor accepts *AStyle* and *ATextLen* parameters.

One of the standard Windows combo box styles (CBS_SIMPLE, CBS_DROPDOWN, or CBS_DROPDOWNLIST) is passed to the constructor of a combo box as the *AStyle* parameter. The *Attr.Style* data member of the combo box is set using the *AStyle* parameter and additional default styles, as shown next:

```
Attr.Style = WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP |
             CBS_SORT | CBS_AUTOHSCROLL | WS_VSCROLL | AStyle;
```

In the following code sample, the *Attr.Style* data member of a combo box is modified so that the combo box's list box will be unsorted:

```
Combo3 = new TComboBox(this, ID_COMBO3, 190, 160, 150, 100,
                       CBS_DROPDOWNLIST, 40);
Combo3->Attr.Style &= ~CBS_SORT;
```

The *ATextLen* parameter is the maximum length of the edit control portion of the combo box.

Modifying combo boxes

TComboBox defines two member functions for showing and hiding the list area of drop down and drop down list combo boxes: **ShowList** and **HideList**. Both are procedures and take no arguments. You do not need to call these member functions to show and hide the list when the user clicks the down arrow to the right of the edit area. This is an automatic response for a Windows combo box. These member functions are useful to force the list to be shown and hidden.

Sample application: CBoxTest

The program *CBoxTest* produces the application shown in Figure 12.4. It uses all three types of combo boxes. *Combo1* is a simple combo box, *Combo2* is a drop down combo box, and *Combo3* is a drop down list combo box.

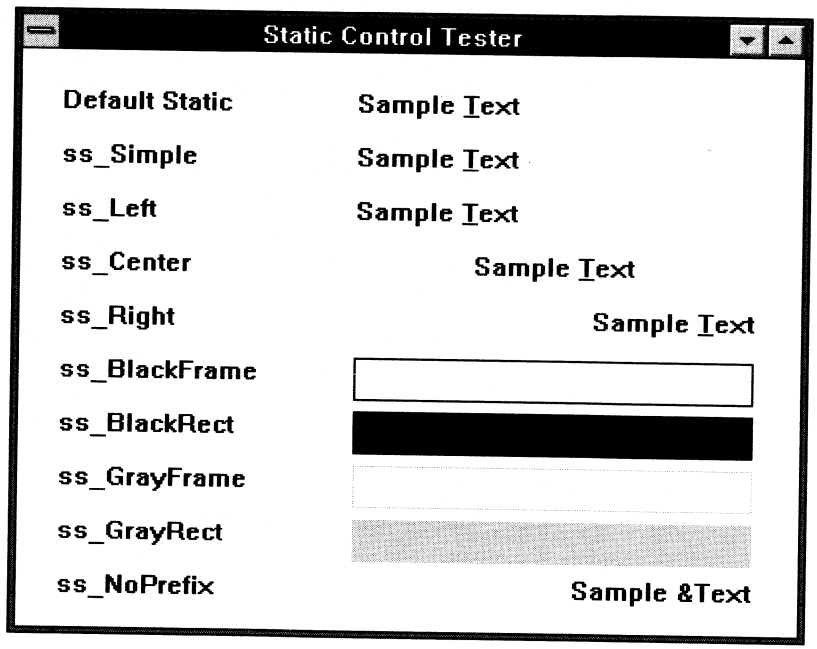
Clicking the Show and Hide buttons forces the list of the bottom right combo box, *Combo3*, to be shown and hidden by calling the member functions **ShowList** and **HideList**.

The complete file, CBOXTEST.CPP, can be found in the EXAMPLES subdirectory.

Static controls

Static controls are usually unchanging text or simple graphics that can appear in a window or dialog. The user does not interact with static controls, although the program can change their text. Figure 12.6 shows a variety of static control styles and their corresponding Windows style constants.

Figure 12.6
Static controls



Constructing static controls

Because static controls don't send notification messages and aren't normally manipulated programmatically through their ID, they don't need to have unique IDs. By convention, the "don't-care" ID for a control, often used for static controls, is –1.

In addition to the usual constructor parameters for a control, **TStatic**'s constructor takes one additional parameter, the text length, which contains the length of the text that **TStatic** objects *transfer*. See "Transferring control data" (the last section of this chapter) for a description of data transfer, and the role of **TStatic**'s *TextLen* data member.

TStatic constructors invoke **TControl**'s constructor, and then add SS_LEFT (for left-justified text) and remove the WS_TABSTOP style in its *Attr.Style* data member. To change the styles set by default for your static control, modify *Attr.Style*:

```
AStatic = new TStatic(this, -1, "&Text", 20, 50, 200, 24, 6);
AStatic->Attr.Style = AStatic->Attr.Style & ~SS_LEFT | SS_CENTER;
```

If you don't need to change your static control's styles after construction, you do not need to maintain a reference to it as a local variable. If, as is usually the case, you don't need to manipulate the static control after it is created, you do not need to maintain a reference to it as a data member of its parent window.

One available option for static controls is to underscore one or more characters in the text string. The implementation and effect of this is similar to underscoring the first character of a menu choice: You insert an & character in the string immediately preceding the character to be underscored. For example, to underscore the *T* in *Text*, send the string "&Text" in the constructor call. If you want the & character in the string, use the Windows static style SS_NOPREFIX (see Figure 12.6).

Querying static controls

To inquire about the current text held by a static control, use the **GetText** member function.

Modifying static controls

TStatic has two member functions for altering the text of a static control. **SetText** sets the static's text to be the passed **LPSTR** argument. **Clear** erases the static's text. However, you cannot change the text of static controls created with the style SS_SIMPLE.

Example: StatTest application

The STATTEST.CPP application creates the static testing application shown in Figure 12.6. Note that the labels, such as "Default Static" and "SS_SIMPLE," are static controls, as are "Sample Text" and the black and gray boxes.

The complete file, STATTEST.CPP, can be found in the EXAMPLES subdirectory.

Edit controls

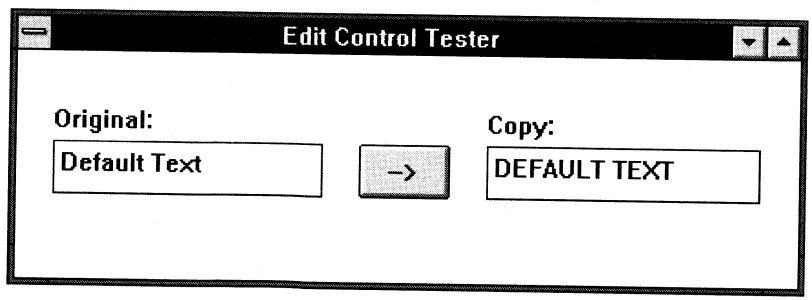
Edit controls are rectangular controls (with or without visible borders) that are used to retrieve text input from a user. An edit control can be "typed into" by a user, and its text can also be modified programmatically. Many operations on an edit control (such as Cut, Copy, and Paste) send and retrieve text to and from the Clipboard. 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)

Windows provides edit controls with the ability to respond to many user events. For example, when you double-click a word in an edit control, the word is selected; when you press *Del*, the character at the cursor position is deleted. ObjectWindows expands on the capabilities of a Windows edit control and provides extensive text-editing behavior.

Figure 12.7 shows a window that contains two edit controls.

Figure 12.7
A window with edit controls



Constructing edit controls

TEdit's constructor accepts the usual control constructor parameters: a pointer to a parent window, a control ID, and location and size data. In addition, the **TEdit** constructor accepts a **BOOL** parameter, *Multiline*, and an *AText* parameter of type **LPSTR**. *Multiline* indicates whether or not a multiline edit control is to be created. Another parameter, *ATextLen*, contains the maximum number of characters that can be entered in the edit control. Because the text must include a terminating null value, the number of characters that can be entered in the edit control is actually one less than the *ATextLen* parameter that is passed.

A single-line and a multi-line edit control are constructed in the following sample code. The user will be able to enter a maximum of 39 characters in either of these edit controls:

```
SingleLineEdit = new TEdit(this, ID_SEDIT, "Default Text",
                           20, 50, 150, 30, 40, FALSE);
MultilineEdit = new TEdit(this, ID_MEDIT, "Default Text",
                          20, 50, 150, 30, 40, TRUE);
```

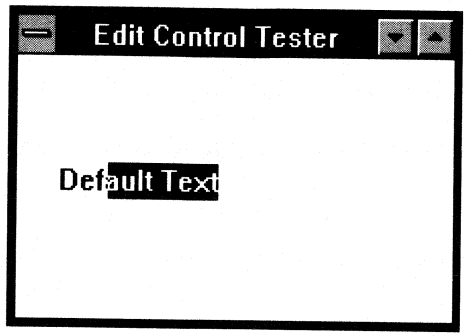
The following default styles are set for all edit controls: WS_CHILD, WS_VISIBLE, WS_TABSTOP, ES_LEFT, ES_AUTOHSCROLL, and WS_BORDER. If *Multiline* is TRUE, the control is a multiline edit control with the additional default styles: ES_MULTILINE, ES_AUTOVSCROLL, WS_HSCROLL, and WS_VSCROLL.

After the edit control has been constructed, its style can be changed:

```
BorderlessEdit = new TEdit(this, ID_SEDIT, "Default Text",
                           20, 50, 150, 30, 40, FALSE);
BorderlessEdit->Attr.Style &= ~WS_BORDER;
```

These lines produce the edit control shown in Figure 12.8 when the **Create** member function of the edit control is called.

Figure 12.8
An edit control created with no border



The following table summarizes some common edit control styles:

Table 12.3
Common edit control styles

Style	Result
ES_LEFT	Left justified
ES_CENTER	Center justified
ES_RIGHT	Right justified
ES_MULTILINE	Multiline edit control
ES_UPPERCASE	Converts text to uppercase characters
ES_LOWERCASE	Converts text to lowercase characters
ES_AUTOVSCROLL	Automatically scrolls multiline edit control text vertically
ES_AUTOHSCROLL	Automatically scrolls multiline edit control text horizontally
WS_BORDER	Adds rectangular box around edit control
WS_VSCROLL	Adds vertical scroll bar
WS_HSCROLL	Adds horizontal scroll bar

Clipboard and editing operations

You can directly transfer text between an edit control and the Windows Clipboard using the following **TEdit** member functions:

- **Cut**
- **Copy**
- **Paste**

Other editing member functions are available:

- **Clear** (derived from **TStatic)**
- **DeleteSelection**
- **Undo**

One additional editing member function available is the **BOOL** member function, **CanUndo**, which determines if the last change can be undone.

Often, you will want to give the user access to these member functions through an editing menu. An edit control automatically responds to menu choices such as Edit | Copy and Edit | Undo. **TEdit** defines command-based member functions, such as **CMEditCopy** and **CMEditUndo**, which are invoked in response to a particular menu selection (command) in the edit control's parent window. **CMEditCopy** calls **Copy** and **CMEditUndo** calls **Undo**.

The following table lists the member functions invoked in response to menu selections with particular menu IDs (defined in rcowl.h):

Table 12.4
Menu IDs and the member functions they invoke

Menu ID	Member function invoked
CM_EDITCUT	**CMEditCut**
CM_EDITCOPY	**CMEditCopy**
CM_EDITPASTE	**CMEditPaste**
CM_EDITCLEAR	**CMEditClear**
CM_EDITDELETE	**CMEditDelete**
CM_EDITUNDO	**CMEditUndo**

All you have to do is add an editing menu whose items have the menu IDs listed here. You need not write any member functions.

Querying edit controls

Often, you will want to query an edit control to validate its text entry, store the entry for later use, or copy the entry to another control. **TEdit** supports a number of querying member functions. Many of the edit control's query and modification member functions return, or require you to specify, a line number or a character's position in a line. All these indexes start at zero. In other words, the first line is line zero and the first character in any line is character zero. The most important query member functions are **GetText**, **GetLine**, **GetNumLines**, and **GetLineLength**.

You will notice that the **TEdit** query member functions that return text from an edit control retain the text's formatting. This is important only for multiline edit controls, which allow text to appear in more than one line. In this case, returned text that spans more than one line in the edit control contains two extra characters for each line break: *carriage return* (0x0D) and *linefeed* (0x0A). When this text is inserted back in an edit control, pasted from the Clipboard, written to a file, or printed to a printer, the line breaks appear as they did in the edit control. Thus, when you use a query member function that gets a specified number of char-

acters, be sure to account for the two extra characters taken up by a line break.

GetText retrieves the text from an edit control. It fills the string pointed to by the passed **LPSTR** argument with the contents of the edit control, including line breaks, up to the number of characters specified in the second parameter. It returns, as the value of the call, the number of characters that were copied. Zero is returned if the edit control contains no text.

The following procedure gets the text from an edit control and returns it in the *TextBuffer* argument:

```
void TTestWindow::ReturnText(LPSTR TextBuffer, int BufferSize)
{
  char TheText[20];

  if (BufferSize >= sizeof(TheText) &&
      (Edit->GetText(TheText, sizeof(TheText) != 0))
    strcpy(TextBuffer, TheText);
  else strcpy(TextBuffer, "")
}
```

GetLine is more specialized than **GetText**. In a multiline edit control, it returns the text in the line specified by the integer argument. Line 0 is the first line.

GetNumLines returns the number of lines entered in a multiline edit control. You should use **GetNumLines** to check how many lines have been entered in an edit control before getting the text from a line with **GetLine**.

GetLineLength returns the number of characters in the specified line in a multiline edit control. If the line exists, but holds no characters, **GetLineLength** returns zero. You should use **GetLine-Length**, **GetNumLines**, and **GetLine** together to safely get the text from a multiline edit control.

```
void TTestWindow::ReturnLine(int LineNum, LPSTR TextBuffer,
                             int BufferSize) {
  char TheText[20];

  if ((Edit->GetNumLines >= LineNum) &&
      (Edit->GetLineLength(LineNum) >= sizeof(TheText)) &&
      (Edit->GetLine(TheText, sizeof(TheText)) != 0))
    strcpy(TextBuffer, TheText);
  else
    strcpy(TextBuffer, "")
}
```

GetSubText is another member function more specialized than **GetText**. It takes as arguments a **LPSTR** and two integers representing the starting and ending indexes—a range—of the edit control's text. The first character is at index zero. In a multiline edit control, the index is counted sequentially from the first line through the remaining lines. Line breaks are counted as two characters. This lets your program reconstitute the format of the edit control's text when displaying it, printing it, transferring it to other edit controls, or placing it in the Clipboard.

You might wonder where you get the indexes to pass as arguments in the **GetSubText** member function call. One way is to use **GetSelection**, a member function that returns the starting and ending index of the text that is currently selected, or highlighted. Usually the text has been selected by the user, but it can also be selected by the program, using **SetSelection** (see the following section, "Modifying edit controls"). The ending index returned by **GetSelection** is the index of the last selected character plus one.

You might be able to save some processing time in an application that involves processing the text from edit controls. The **IsModified** member function, a **BOOL** function, returns TRUE if the user has modified the edit control's text and FALSE if it is intact. A return of FALSE might save you from calling **GetText** or **GetLine**. **ClearModify** resets the change flag to FALSE.

GetLineIndex and **GetLineFromPos** are two member functions you can use to compute the placement of text in a multiline edit control only. **GetLineIndex** returns the number of characters (including two for each line break) in all the lines preceding the line specified by the integer argument. The member function returns all the edit's characters if the specified line does not exist. **GetLineFromPos** takes an index to a character position and returns the corresponding line number. Both of these member functions are useful for querying the line structure and contents of a multiline edit control.

Modifying edit controls

In a traditional data entry program, you might not need your program to directly modify an edit control. The user modifies the text and the program reads the text with a query member function. However, many other uses of edit controls require that your application explicitly substitute, insert, clear, or select text. In an ObjectWindows application, the function members of instances

of the **TEdit** class can be called to programmatically manipulate the text of an edit control.

Deleting text

Clear simply deletes the entire text entry in an edit control. You can use it as part of a resetting operation that clears some or all the edit controls in a window. **DeleteSelection** deletes the currently selected text of an edit control. A full text editor might provide access to **Clear** and **DeleteSelection** from menu choices (see "Clipboard and editing operations" on page 171). **DeleteSelection** is a **BOOL** function that returns FALSE if there is no text currently selected.

DeleteSubText is similar to **DeleteSelection** except it deletes the text between the passed starting and ending positions. The removed text includes the character at the starting position but not the character at the ending position.

DeleteLine deletes all text found at the specified line. It does not, however, delete the two characters that make up the line break. Thus, no other line is affected.

Inserting text

Insert is similar to **Paste** except that it gets the text to insert from the supplied argument rather than from the Clipboard. It deletes any existing selected text but does not highlight the inserted text. Using **Insert**, you can implement a text-only Clipboard under control of your program.

SetText performs the combined actions of **Clear** and **Insert**. It deletes the entire contents of the edit control and inserts the text in the supplied argument. For example, to reset an order entry screen for an order entry system that gets most of its orders from California, you might call

```
StateField->SetText("CA");
```

to reset the *state* entry field.

Forcing text selection and scrolling

SetSelection forces the selection, or highlighting, of text between the passed starting and ending positions, not including the character at the ending position.

Scroll forces scrolling in a multiline edit control. **Scroll** takes two integer arguments: the number of characters to scroll horizontally and the number of lines to scroll vertically. A positive integer scrolls to the right, or down, and a negative number scrolls to the left, or up. Only multiline edit controls can scroll vertically.

Sample program: EditTest

EditTest is a program that displays a main window that serves as the parent window for two edit controls, two static controls, and a button. This window is depicted in Figure 12.7 on page 169.

When the user clicks the button, the text from the left edit control (*Edit1*) is copied to the right edit control (*Edit2*). The text is converted to uppercase in *Edit2* because the ES_UPPERCASE style was added to the default creation styles in its *Attr.Style* data member. If no text is selected in *Edit1*, all its text is copied to *Edit2*. If some text is selected in *Edit1*, only the selected text is copied.

The edit menu supports editing functions in whichever edit control currently has the input focus.

The complete file, EDITTEST.CPP can be found in the EXAMPLES subdirectory.

Push button controls

Push buttons (sometimes called "command buttons") are used to perform some task each time the button is pressed. There are two styles of push buttons, both derived from **TButton**: BS_PUSHBUTTON and BS_DEFPUSHBUTTON. Either style can be used for instances of **TButton**. Default push buttons are similar to push buttons but have a bold border and are generally used to indicate the default user response.

TButton's constructor accepts the usual control constructor parameters: a parent window, a control ID, and location and size data. In addition, the **TButton** constructor accepts an *IsDefaultButton* **BOOL** parameter, and a *Text* parameter of type **LPSTR**. *IsDefaultButton* indicates whether or not the button is a default push button. *Text* contains the text to be displayed on the button. For example, the following statement from the constructors of the button's parent constructs a button with a caption "Test Button."

```
Push1 = new TButton(this, ID_BUTTON, "Test Button",
                    38, 48, 316, 24, FALSE);
```

Responding to button messages

Whenever the user clicks a push button, the button's parent window receives a notification message from the button. The parent window can respond to these messages by defining a child-ID-based response member function.

In the following example, **TTestWindow::HandleButtonMsg** responds to notification messages from a button with an ID_BUTTON identifier by putting up a message box:

```
class TTestWindow : public TWindow {
public:
  PTRadioButton Button;
  virtual void HandleButtonMsg(RTMessage Msg) =
    [ID_FIRST + ID_BUTTON];

void TTestWindow::HandleButtonMsg(RTMessage Msg)
{
  MessageBox(HWindow, "clicked", "The button was:", MB_OK);
}
```

The only notification code defined by Windows for push buttons is BN_CLICKED, so the code doesn't need to be checked.

Check boxes and radio buttons

Check boxes and radio buttons are sometimes collectively referred to as *selection boxes* because they are both used to retrieve user input in the form of a selection. Check boxes and radio buttons normally have two states, *checked* and *unchecked*. Special three-state check boxes have an additional *grayed* state. When a check box is *grayed*, the selection it represents is disabled and the box cannot be checked.

Normally, check boxes are used to retrieve options (perhaps multiple) from a user, and radio buttons are used to retrieve one of several mutually exclusive options. For example, check boxes might be used to allow a user to pick a number of fonts to be loaded. Radio buttons could be used to retrieve a font type for a particular character.

Check boxes and radio buttons are represented by instances of the

ObjectWindows **TCheckBox** and **TRadioButton** classes. **TCheckBox** is derived from **TButton**, and **TRadioButton** is derived from **TCheckBox**.

Constructing check boxes and radio buttons

TCheckBox and **TRadioButton** constructors accept the usual control constructor parameters: a parent window, a control ID, and location and size data. In addition, both constructors accept a a *Text* parameter of type **LPSTR** and an *AGroup* parameter of type **PTGroupBox**. *Text* contains the text which, by default, will be displayed to the right of the selection box. *AGroup* is a pointer to a **TGroupBox** object that is used to aid the logical grouping of selection boxes. If *AGroup* is NULL, the selection box is not part of any logical group (see "Group boxes" on page 180):

```
GroupBox = new TGroupBox(this, ID_GROUPBOX, "A Group Box",
                         38, 102, 176, 108);
CheckBox = new TCheckBox(this, ID_CHECKBOX, "Check Box Text",
                         235, 12, 150, 26, GroupBox);
```

By default, BS_AUTOCHECKBOX and BS_AUTORADIO-BUTTON styles are set by the **TCheckBox** and **TRadioButton** constructors. As a result, Windows automatically toggles the check state of either type of selection box when it is clicked. When a radio button with the "auto" style is clicked, Windows additionally unchecks all other radio buttons in the same group. If you reset *Attr.Style*, specifying "nonauto" BS_CHECKBOX or BS_RADIOBUTTON styles, you'll be responsible for managing the check states of your selection boxes.

If you'd like the text strings of your check boxes and radio buttons to appear to the left of the box, add the BS_LEFTTEXT style to the *Attr.Style* before you create them:

```
CheckBox->Attr.Style |= BS_LEFTTEXT;
```

Querying the state of a selection box

Querying a selection box is one way to find out and respond to its state. Radio buttons and check boxes have two states: checked and unchecked. Use the **GetCheck** member function of **TCheckBox** to query the state of a selection box:

```
MyState = CheckBox->GetCheck();
```

The return value of **GetCheck** can be compared with the defined constants BF_UNCHECKED, BF_CHECKED, and BF_GRAYED to determine the state of the box.

Modifying the state of a selection box

Modifying (by checking and unchecking) a selection box's state sounds like a job for your program's user, not you. But in some cases, your program needs direct control over a selection box's state.

For example, one case in which you might want to control a selection box's state is to display options that have previously been selected and saved. The **TCheckBox** class defines four member functions for modifying a check box's state: **Check**, **Uncheck**, **Toggle**, and the most general, **SetCheck**.

- **Check** forces the check box's state to be checked:

```
CheckBox->Check();
```

- **Uncheck** forces the check box's state to be unchecked:

```
CheckBox->Uncheck();
```

- **Toggle** changes the check box's state from checked to unchecked, or vice versa. In the case of three-state boxes, **Toggle** changes an unchecked box to checked, a checked box to grayed, and a grayed box to unchecked.

```
CheckBox->Toggle();
```

- **SetCheck** gives you complete control over the state of the check box:

```
CheckBox->SetCheck(BF_UNCHECKED);   // Unchecks selection box
CheckBox->SetCheck(BF_CHECKED);     // Checks selection box
CheckBox->SetCheck(BF_GRAYED);      // Grays a three-state
                                       selection box
```

When a radio button with the BS_AUTORADIOBUTTON is checked by a call to one of these member functions, any checked radio buttons in the same group will be unchecked (see "Group boxes" on page 180). By convention, one and only one of the radio buttons in a group should always be checked.

Responding to check box and radio button messages

When a user selects a check box or radio button, a notification message with a BN_CLICKED notification code is sent to its parent window. As is the case for instances of the other control classes, these messages are normally handled in child-ID-based response member functions of the parent window. You may, however, want to derive classes from **TButton** or **TCheckBox**, whose instances perform some action when pressed.

Group boxes

A Windows group box element is a labeled rectangle that visually associates other controls, and does nothing more. The logical grouping of controls (upon which the mutual exclusivity of auto radio buttons depends) is not a result of their location within a group box element, but is dependent on the order of creation of the controls and the use of the WS_GROUP creation style.

ObjectWindows **TGroupBox** objects, however, are no "dummies." They perform a useful function in the logical grouping of controls. By specifying a **TGroupBox** for your selection boxes (when constructed), you'll be able to define a "group" response method rather than having to define a separate response method for each. **TGroupBox** makes this possible by accepting notification messages from the selection boxes in its group, and translating the message into a "group" notification message to the parent.

Constructing a group box

The TGroupBox constructor accepts the usual control constructor parameters: a parent window, a control ID, and location and size data. In addition, a *Text* parameter of type **LPSTR**, which will become the label of the group box, is passed:

```
GroupBox = new TGroupBox(this, ID_GROUPBOX, "A Group Box",
                         38, 102, 176, 108);
```

Responding to group box messages

When a change is made in the check state of a selection box that is part of a group, its **TGroupBox** is notified. The **TGroupBox** then sends a "group" notification message to its parent. The parent can respond to the message by defining a child-ID-based response member function to handle notification messages from the group box (specifying the identifier of the group box in the extension).

In the GBOXTEST.CPP sample program, a parent window member function is defined to respond to selection changes in each of the groups of radio buttons. GBOXTEST.CPP is installed in your EXAMPLES directory.

Example program: BtnTest

BtnTest is a program that creates a window with push button, check box, radio button, and group box controls. When the application is started, the controls are displayed in the main window. When the user clicks the controls, the application responds in a variety of ways. See Figure 12.9.

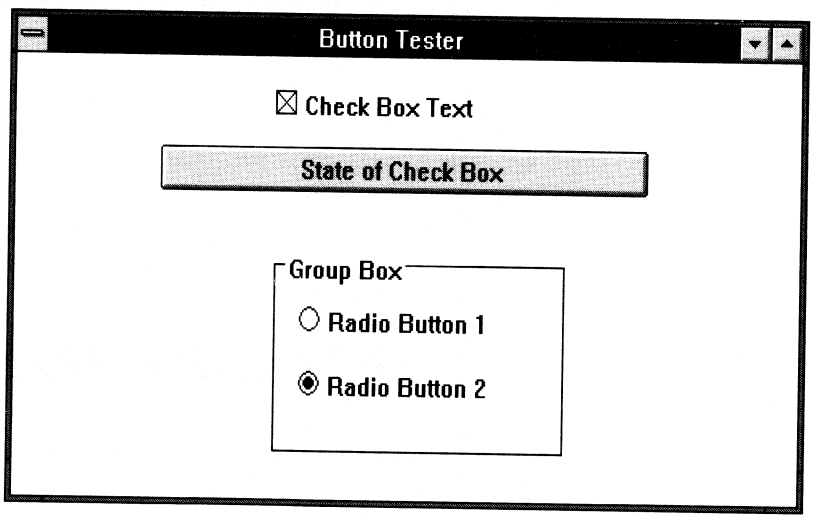


Figure 12.9
A window with various buttons

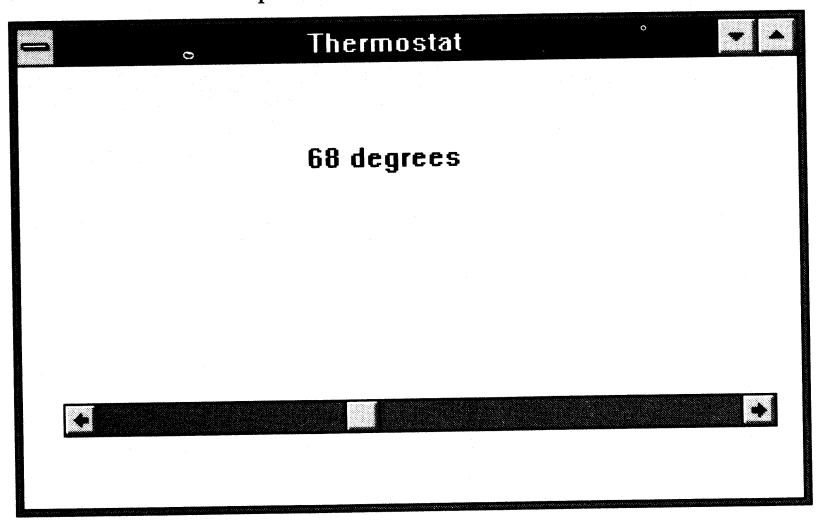
The complete file, BTNTEST.CPP, can be found in the EXAMPLES subdirectory.

Scroll bars

Scroll bar controls contain a sliding box (a thumb), which can be moved along the length of the scroll bar to various positions. Each position is associated with a value in a specified range. Scroll bar controls are used to retrieve a value that has a set range from the user. For example, a scroll bar control could be used to retrieve a temperature setting, or a color value, from a user.

Scroll bar controls are very similar in appearance to, and at times almost indistinguishable from, window scroll bars. Window scroll bars, which control the scrolling of the view of a window, are part of the window itself; they aren't child windows. Windows creates window scroll bars for windows that specify the WS_HSCROLL or WS_VSCROLL creation styles. In ObjectWindows, scroll bar controls are created the same way that the other types of controls are created. All you need to do is construct a scroll bar object in the constructor of its parent.

Figure 12.10
A scroll bar control



Constructing scroll bar objects

The **TScrollBar** constructor accepts the usual control constructor parameters: a pointer to a parent window, a control ID, and location and size data. In addition, an *IsHorizontal* **BOOL** parameter is passed that indicates the orientation (horizontal or vertical) of the scroll bar in its parent window. If a width of zero is supplied for a vertical scroll bar, it is constructed with a standard

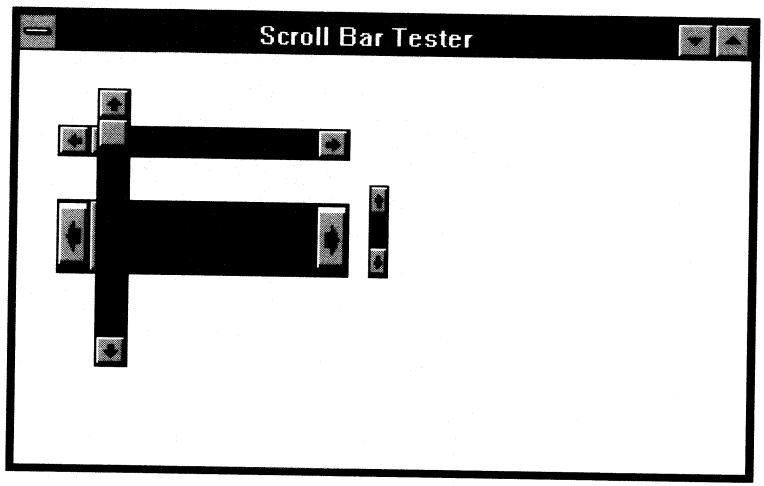
width, like that of a list box. The same is true if a height of zero is specified for a horizontal scroll bar. The following code creates the standard-height horizontal scroll bar shown in Figure 12.10:

```
Thermometer = new TScrollBar(this, ID_THERMOMETER,
                             20, 170, 340, 0, TRUE);
```

The **TScrollBar** constructor invokes the **TControl** constructor and then adds SBS_HORZ or SBS_VERT to the default styles set for the scroll bar. You can specify additional styles, such as SBS_TOPALIGN, by changing the scroll bar's *Attr.Style* data member.

Figure 12.11 shows a variety of scroll bar controls.

Figure 12.11
A window with a variety of scroll bars



The range of the scroll bar is set by default from 1 to 100 when the scroll bar is created. Use the **SetRange** member function, as described in the section "Modifying scroll bars" on page 184, to set the range differently.

Two other attributes of a scroll bar object are its line magnitude and page magnitude. *LineMagnitude* is the number of positions the thumb will be moved when the user clicks the arrow buttons at either end of the scroll bar. *PageMagnitude* is the number of positions the thumb will be moved when the user clicks in the scrolling areas—the spaces between the thumb and either end of the scroll bar. *LineMagnitude* and *PageMagnitude* are set by default to 1 and 10, respectively. You can reset these values by directly modifying the *LineMagnitude* and *PageMagnitude* data members of **TScrollBar**.

Querying scroll bars

TScrollBar defines two member functions for querying a scroll bar: **GetRange** and **GetPosition**. The **GetRange** member function is a procedure that takes two integer references. The procedure fills these integers with the highest and lowest thumb positions in the scroll bar's range.

GetPosition is a function that returns the integer position of the thumb. Often your program will get the range and the position, and then compare the two.

Modifying scroll bars

Modifying scroll bars seems like a job for the user of your programs, and most often it is. However, your program can also modify a scroll bar.

SetRange is a member function that takes two integer arguments for the lowest and highest positions in the range. As a default, new scroll bars have a range from 1 to 100. You might want to change this range to better map the actual entity the scroll bar controls. To do so, call the scroll bar's **SetRange** member function after the scroll bar is created. A good place to do this is in the **SetupWindow** member function of its parent. Be sure that, before you call **SetRange**, you call the **SetupWindow** member function of the parent's base class to create the scroll bar control.

In the following example, the range of a scroll bar used as a thermostat is modified in its parent's **SetupWindow** member function:

```
void TParentWindowType::SetupWindow()
{
  TWindow::SetupWindow();
  Thermometer->SetRange(32, 120);
}
```

Of course, if your scroll bar is an instance of a class that you derive from **TScrollBar**, a better place to modify its range would be in its own **SetupWindow** member function.

SetPosition is a member function that takes one integer argument; the position to which you want to move the scroll bar's thumb. In the thermostat application discussed earlier, your

program could directly set the temperature setting to 78 degrees with

```
Thermometer->SetPosition(78);
```

The third member function, **DeltaPos**, moves the scroll bar's thumb position up (left) or down (right) by the amount specified by the integer argument. A positive integer moves the thumb down (right). A negative integer moves it up (left). For example, to lower the thermostat's temperature setting by 5 degrees, use

```
Thermometer->DeltaPos(-5);
```

Responding to scroll bar events

When a scroll bar is scrolled, its parent window receives a scroll bar notification message. If you want your window to respond to scrolling events, respond to the notification messages in the usual way, by defining child-ID-based response member functions. However, scroll bar notification messages are slightly different than the other control notification messages. They are based on the Windows messages WM_HSCROLL and WM_VSCROLL, rather than on WM_COMMAND. The only difference you must be aware of is that the scroll bar notification codes are stored in *Msg.WParam*, rather than in *Msg.LP.Hi*. Some common codes include SB_LINEUP, SB_LINEDOWN, SB_PAGEUP, SB_PAGEDOWN, SB_THUMBPOSITION, and SB_THUMBTRACK.

Most often, you'll respond to all scroll bar events in the same manner: by retrieving the current position and taking appropriate action. You'll usually ignore the notification code, as in the following example:

```
void TTestWindow::HandleThermMsg(RTMessage)
{
  int NewPos;

  NewPos = Thermometer->GetPosition();
  // Do some processing based on NewPos
}
```

You may not want to respond to a thumb drag until the user has chosen a new location. In that case, screen out the messages with the SB_THUMBTRACK code.

```
void TTestWindow::HandleThermMsg(RTMessage)
{
```

```
  int NewPos;

  if (Msg.WParam != SB_THUMBTRACK)
  {
    NewPos = Thermometer->GetPosition();
    // Do some processing based on NewPos
  }
}
```

Occasionally you may want to have the scroll bar object itself respond to the scroll bar notification message, building a particular response behavior into the scroll bar object. To program a scroll bar object to directly respond to its notification messages, define a notify-based response member function for its class. Use the sum of NF_FIRST and the scroll bar notification code as the virtual method declaration. To do this, derive a new class from **TScrollBar**:

```
class TSpecializedSBar : public TScrollBar
{
public:
  virtual void SBTop(RTMessage Msg) =
    [NF_FIRST + SB_TOP];
};

TSpecializedSBar::SBTop(RTMessage Msg)
{
  TScrollBar::SBTop(Msg);
  // Do special handling when scroll bar is positioned to top
}
```

Note that the member function **SBTop** first calls **TScrollBar::SBTop**. **TScrollBar::SBTop** moves the scroll bar's thumb when an SB_TOP notification message is received. **TScrollBar** defines member functions to respond to the corresponding notification message:

Table 12.5
Member functions defined by TScrollBar

TScrollBar member function	Notification Message
SBLineUp	SB_LINEUP
SBLineDown	SB_LINEDOWN
SBPageUp	SB_PAGEUP
SBPageDown	SB_PAGEDOWN
SBThumbPosition	SB_THUMBPOSITION
SBThumbTrack	SB_THUMBTRACK
SBTop	SB_TOP
SBBottom	SB_BOTTOM

If you redefine any of these member functions, call the **TScrollBar** member function that you redefined to move the scroll bar's thumb.

Example: SBarTest

The *SBarTest* program creates the thermostat application shown in Figure 12.10.

The complete file, SBARTEST.CPP, can be found in the EXAMPLES subdirectory.

Transferring control data

To manage complex dialog boxes or windows with many child window controls, you might typically create a descendant class to store and retrieve the state of its 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. As an alternative, you can avoid defining a descendant object by defining and supplying a structure to represent the state of a window's or a dialog's controls.

For example, your program can bring up a modal dialog box and, after it is closed, extract information from the transfer buffer about the state of each control. If the user brings up the dialog box again, and the transfer buffer has not been modified, the controls will be set to their states when the dialog last closed. In addition, you can set the initial state of each control by initializing the transfer buffer data. You can also explicitly transfer data in either direction at any time, such as to reset the states of controls to their previous values.

Defining a transfer buffer

The transfer buffer is a structure with one field for each control participating in the transfer. There are some controls that are not affected by the transfer mechanism. For instance, push buttons, which have no states, do not participate in transfers. Neither do group boxes.

To define a transfer buffer, define a data member for each participating control in the dialog or window. It is not necessary to define transfer data members for every control in a dialog or

window, only those which you want to transfer values to and from. This transfer buffer stores transfer data for one of each type of control that transfers data.

```
struct SampleTransferStruct {
  char StaticText[STATICTEXTLEN];       // static text
  char EditText[EDITTEXTLEN];           // edit text
  TListBoxData ListBoxData;             // list box strings and
                                           selection
  TComboBoxData ComboBoxData;           // combo box strings and
                                           selected string
  WORD CheckState;                      // check box state
  WORD RadioState;                      // radio button state
  TScrollBarData ScrollBarStruct;       // scroll bar range, etc.
}
```

As you can see, the type of control determines the type of data member defined for the transfer buffer. That's because each type of control transfers different data:

- Static and edit controls transfer their text. Their *TextLen* data members contains the length of the transferred text, including the terminating null.

  List boxes transfer the strings in their lists, the selected string or strings, and the number of selected strings using an object of type **TListBoxData**. You can include a pointer to a **TListBoxData** object in your transfer buffer like this:

```
struct MyTransferBuffer
{
:
  PTListBoxData pMyListBoxData;
:
};
```

  Set up *pMyListBoxData* as follows:

```
MyTransferBuffer.pMyListBoxData = new TListBoxData();
```

  The **TListBoxData** class definition includes three data members:

Table 12.6
TListBoxData data members

Data member	Definition
PArray *Strings*	An array of **String**s (from the container class library) containing the items in the list box
PArray *SelStrings*	An array of **String**s containing the items in the list box. In a single selection list box, there will only be one **String**.
int *SelCount*	The number of selected items. *SelCount* is one for a single selection list box.

- Combo boxes transfer the strings in the list box portion of the combo box, and a pointer to the selected string in an object of type **TComboBoxData**. You can include a pointer to a **TComboBoxData** object in your transfer buffer as you did for list boxes, using **TComboBoxData** instead of **TListBoxData**:

```
struct MyTransferBuffer
{
:
PTComboBoxData pMyComboBoxData;
};
```

Set up *pMyComboBoxData* as follows:

```
MyTransferBuffer.pMyComboBoxData = new TComboBoxData();
```

The **TComboBoxData** class definition includes three data members:

Table 12.7
TComboBoxData data members

Data member	Definition
PArray *Strings*	An array of **String**s (from the container class library) containing the items in the list box part of the combo box.
Pchar *Selection*	A pointer to the character string that is the selected item in the combo box's list box.

- Check boxes and radio buttons transfer their check states as **WORD** values, (BF_UNCHECKED, BF_CHECKED, or BF_GRAYED).
- Scroll bars use another record, a **TScrollBarData**, to store the range and position of the scroll bar control. Here is the definition of **TScrollBarData**:

```
struct TScrollBarData {
   int LowValue;
   int HighValue;
   int Position;
}
```

Constructing controls and enabling transfers

A window or dialog that uses the transfer mechanism must construct its participating control objects in the order in which their corresponding transfer buffer data members are defined. To enable the transfer mechanism for a window or dialog object, simply set its *TransferBuffer* data member to point to a transfer buffer you define.

Associating control objects with control elements is described in Chapter 11, "Dialog objects."

The transfer mechanism requires the use of ObjectWindows objects to represent the controls for which you would like to transfer data. To associate objects with controls in dialog boxes, use the constructor that accepts simply a parent window and a resource identifier as arguments.

```
TParentWindow::TParentWindow(PTWindowsObject AParent, LPSTR ATitle) :
                                TWindow(AParent, ATitle)
{
  TheDialog = new TDialog(this, ID_DIALOG);
  new TStatic(TheDialog, ID_STATIC, 20);
  new TEdit(TheDialog, ID_EDIT);
  new TListBox(TheDialog, ID_LISTBOX);
  new TComboBox(TheDialog, ID_COMBOBOX, 20);
  new TCheckBox(TheDialog, ID_CHECKBOX, 0);
  new TRadioButton(TheDialog, ID_RADIOBUTTON, 0);
  new TScrollBar(TheDialog, ID_SCROLLBAR);
  TheDialog->TransferBuffer = &TransferStruct; }
```

By default, instances of TStatic do not transfer their contents even in dialogs.

Another difference between window and dialog data transfer is that the transfer mechanism is initially disabled by default for a window's controls and enabled by default for a dialog's controls (except for instances of **TStatic**). To enable transfer for a window's control, call its **EnableTransfer** member function.

```
⋮
Edit = new TEdit(this, ID_EDIT, "", 10, 10, 100, 30, 40, FALSE);
Edit->EnableTransfer();
⋮
```

To explicitly exclude a control from the transfer mechanism, call its **DisableTransfer** member function.

Transferring the data

Upon window or dialog creation or dialog execution, data is automatically transferred from the transfer buffer to the set of participating controls.

For a modal dialog only, data is automatically transferred out of the controls and into the transfer buffer when the dialog receives a command message with a control ID of IDOK. Data is automatically transferred out of the controls of a modal dialog and into its transfer buffer when the call in **CloseWindow** to **CanClose** returns TRUE. Then, if the dialog is executed again, the buffer data is transferred again to the controls. Under this scheme, the dialog and the buffer remain in synch.

However, 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. 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 button. Use the **TransferData** member function in either case, supplying the TF_SETDATA constant to transfer from the buffer to the controls and the TF_GETDATA constant to transfer in the other direction. For example, you might want to call **TransferData** in the **CloseWindow** member function of a window object:

```
void TSampleWindow::CloseWindow()
{
  TransferData(TF_GETDATA);
  TWindow::CloseWindow();
}
```

Customizing transfer for controls

You may want to modify the way a particular control transfers its data, or include a new control you define in the transfer mechanism. In either case, you simply need to write a **Transfer** member function for your control object that, when TF_GETDATA is passed, copies data from the control to the location whose address is passed. If TF_SETDATA is passed, simply copy the data at the passed address into the control. If TF_SIZEDATA is passed, just return the size of the transfer data, without doing anything. The size of the transfer data is available in **TStatic**'s *TextLen* data member. As an example, here is **TStatic::Transfer**:

```
WORD TStatic::Transfer(Pvoid DataPtr, WORD TransferFlag)
{
  if (TransferFlag == TF_GETDATA)
    GetText((LPSTR)DataPtr, TextLen);
  else
    if (TransferFlag == TF_SETDATA)
      SetText((LPSTR)DataPtr);
  return TextLen;
}
```

The **Transfer** member function should always return the number of bytes transferred.

Transfer examples

The **TranTest** program's main window produces a modal dialog with fields for the user to enter name and address information. It uses a transfer buffer to store this information and display it in the dialog's controls when the dialog is again executed. Notice that a new dialog class didn't need to be defined to set and retrieve the dialog's data.

TRANTEST.CPP can be found in the EXAMPLES subdirectory. The LBXTTEST.CPP and CBXTTEST.CPP sample programs demonstrate transfer of list box and combo box data. These programs are modified versions of the LBOXTEST.CPP and CBOXTEST.CPP programs presented earlier in this chapter.

C H A P T E R

13

Customizing control objects

The default characteristics, appearance, and behavior of the ObjectWindows control classes have been presented in the preceeding chapters. Some of the ways you can modify these defaults have been mentioned. This section describes, in detail, the techniques you can use to modify the characteristics of a predefined ObjectWindows control. The procedure for designing your own custom control is also presented.

Modifying a predefined control

There are two basic ways to modify the characteristics of a predefined ObjectWindows control:

- modify its creation styles
- define message response methods

Modifying creation styles

You can modify some default behavior of a predefined control by changing its style creation attribute (which *Attr.Style* contains). For example, to prevent mutually exclusive selections in a group of radio buttons, remove BS_AUTORADIOBUTTON from their creation styles, and add BS_RADIOBUTTON. (Note that you'll now need to check the buttons yourself.) As another example, you can allow multiple items to be selected in a list box by adding LBS_MULTIPLESEL to the creation styles of the list box.

To a limited extent, you can modify the appearance of a predefined control by changing its creation styles. For example, to specify that the text of a check box control be displayed to the left of the box:

```
MyBox = new TCheckBox(TheParent, ID_CHECKBOX, "Text",
                      100, 100, 80, 20, NULL);
MyBox->Attr.Style |= BS_LEFTTEXT;
```

You can add a border to many controls by adding WS_BORDER to their creation styles.

You can specify an *owner-draw* creation style for some predefined controls for greater control over a control's appearance. *Drawable* controls are discussed in the next section.

Modifications that you can make to the characteristics of a predefined control by changing creation styles are limited to the style options available for the control's type. See the online Help for a complete list of the optional creation styles for each type of control.

F1
Help

Making a predefined control drawable

If you'd like to define the appearance of your control (within its predetermined shape), you can specify an owner-draw style for certain types of predefined controls. To make a button drawable, for example, specify BS_OWNERDRAW as one of its creation styles:

```
Attr->Style |= BS_OWNERDRAW;
```

In addition, you'll need to define the manner in which it is to be drawn. To do so, define a **ODADrawEntire** member function in a derived class. You can optionally redefine **ODAFocus** and **ODASelect** member functions to modify the appearance of your drawable control when it receives the focus or when it is selected.

ODADrawEntire, **ODAFocus**, and **ODASelect** member functions are passed a **DRAWITEMSTRUCT** that contains information used for drawing a control. Its *hDC* data member contains a handle to the display context to be used; this handle must be passed to the Windows GDI functions called when drawing the control. Its *rcItem* data member contains the client area of the control.

See the online Help for a detailed description of ***DRAWITEMSTRUCT****.*

A drawable control can also be drawn by a parent window that redefines **WMDrawItem**. A pointer to a **DRAWITEMSTRUCT** is passed as the *LParam* of the specified **RTMessage**; the identifier of the control to be drawn is passed in the *CtlID* member of the structure.

The DCTLTEST.CPP application in the EXAMPLES directory displays a dialog with drawable button controls when a "Test" menu item is selected. Bring up the dialog to see how the drawable buttons respond to focus and selection changes. Use *Tab* and the arrow keys to shift the focus, and click the buttons to select them. In your own applications, you might want to use bitmaps for your drawable buttons.

Push buttons, check boxes, radio buttons, and group boxes can be made drawable by specifying BS_OWNERDRAW as a creation style. List boxes and combo boxes can be made drawable by specifying LBS_OWNERDRAW and CBS_OWNERDRAW, respectively, as creation styles. Windows does not provide this sort of flexibility for scroll bars, edit controls, and static controls.

Modifying predefined message responses

You can define a custom response to any Windows message that a predefined control receives. To do this, define message response member functions in a class derived from the predefined control class. You may want to experiment with using this technique to modify the appearance or behavior of a predefined control.

Specifying additional processing for a predefined control

You can perform additional processing for a incoming message to a predefined control, before or after calling **DefWndProc** to invoke the predefined response.

For example, to have a button beep when clicked in addition to notifying its parent, redefine **WMLButtonDown**:

```
void TBeepButton::WMLButtonDown(RTMessage Msg)
{
  MessageBeep(0);
  DefWndProc(Msg);
}
```

Since default processing of WM_LBUTTONDOWN for a button is to send a BN_CLICKED notification message, an alternate implementation would be to redefine **BNClicked**.

As another example, you could define **WMSetFocus** and **WMKillFocus** member functions in order to modify the border of an edit control when it has the focus. Since the default processing when these messages sends **ENSetFocus** and **ENKillFocus** notification messages, an alternate implementation would be to define **ENSetFocus** and **ENKillFocus** member functions.

Overriding a predefined control's response

Within limits, you may want to override the default processing for a predefined control. There may be a lot more to this than you'd think, since default processing for controls can be complex. Be sure to thoroughly test the results. At times, you'll have to simulate certain predefined behavior.

For example, to implement a read-only edit control, you could make a "do-nothing" **WMChar** response in a class derived from **TEdit**. Your edit control could then be used to display text that could be copied to the Clipboard, but which could not be modified.

As another example of overriding predefined behavior, you may want to override the predefined processing of **WMLButtonDown** for a button to implement a "flat" button that doesn't modify its appearance when clicked. You'll then have to send a notification message with a BN_CLICKED code to the button's parent yourself, because it is normally sent in the predefined processing of **WMLButtonDown**.

```
void TFlatButton::WMLButtonDown(RTMessage Msg)]
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, BN_CLICKED));
}
```

If you find yourself customizing a lot of responses, it's time to start from scratch by defining your own custom control.

Using a custom control

If the behavior you require for a control does not closely match that of a predefined control, you'll need to use a custom control. You define the appearance of a custom control, as well as all of its control behavior.

Designing a custom control

The basic steps to designing a custom control are to:

- determine the types of user input that is to be retrieved
- define the visual appearance of your control
- determine the responses to input

A simple control commonly responds to a left-button mouse click by sending a notification message to its parent. You're free to design your control to retrieve and respond in any way that works for your application. But, remember, your goal is to provide an easy-to-use and consistent interface, and that may mean following some conventions.

Defining a custom control

The basic steps to implementing a custom control are to

- declare a class derived from **TControl**
- redefine **Paint** to paint your control
- define message response methods that intercept and respond to user input

A custom control is used in the CCTLTEST.CPP sample application supplied in the EXAMPLES directory. This color control represents a color option that can be selected by the user. A **TColorControl** object responds to left-button clicks by sending a notification message to its parent:

```
void TColorControl::WMLButtonDown(RTMessage)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, CN_CLICKED));
}

void TColorControl::WMLButtonDblClk(RTMessage)
{
  SendMessage(Parent->HWindow, WM_COMMAND, Attr.Id,
              MAKELONG(HWindow, CN_DBLCLICKED));
}
```

The CN_CLICKED and CN_DBLCLICKED constants passed to the parent of the color control are notification codes that describe the type of event (single- or double-click) that occurred. Custom controls send notification messages with custom notification codes; a custom control can send any code its parent understands.

Since these control notifications are nonstandard, you will have to define them yourself:

```
const unsigned int CN_CLICKED = 0;
const unsigned int CN_DBLCLICKED = 1;
```

The TColorControl in the *CCTLTEST* application redefines **Paint** to paint its client area using a brush with the color it represents. You'll also need to redefine **Paint** for your custom control; after all, an invisible control is hardly useful! You may also want to provide **WMSetFocus** and **WMKillFocus** message response member functions to change your control's appearance when it receives and loses the focus.

*See "Window class registration" in Chapter 10, "Window objects," for more information on these default attributes, and on redefining **GetWindowClass** and **GetClassName**.*

TControl inherits **TWindow**'s **GetWindowClass** member function that specifies the registration attributes that are appropriate, as defaults, for a custom control. If you want to modify these defaults for your custom control, you'll need to redefine **GetWindowClass** and **GetClassName**. **TColorControl**, for example, modified the registration class style to specify that the control was to receive double-click messages:

```
void TColorControl::GetWindowClass(WNDCLASS& AWndClass)
{
  TControl::GetWindowClass(AWndClass);
  AWndClass.style |= CS_DBLCLKS;
}

LPSTR TColorControl::GetClassName()
{
  return "ColorControl";
}
```

You'll need to specify the Windows class name of a dialog's custom controls in the dialog's resource definition. The Windows class name of a control is the name returned by its **GetClassName** member function. If you haven't redefined **GetClassName**, it will return "OWLWindow."

C H A P T E R

14

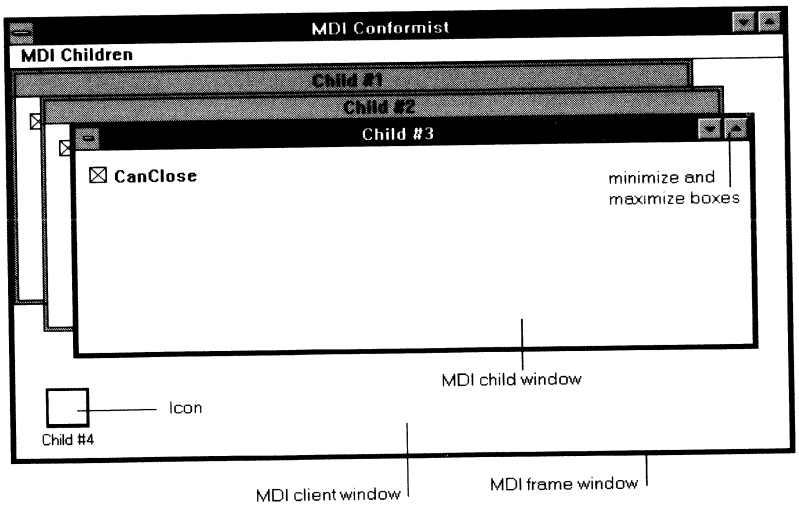
MDI objects

The multiple document interface (MDI) is an interface standard for Windows applications that allows the user to simultaneously work with many open documents. A document, in this sense, is usually a file-specific task, such as editing a text file or working on a spreadsheet file. In MDI-compliant applications, the user can, for example, have many open files within one application. Examples of MDI applications you might have used include the Windows Program Manager and the Windows File Manager. The MDI standard is also part of IBM's Common User Access specification.

The components of an MDI application

There are certain components that are present in every MDI application. Most evident, there is the main window called the *MDI frame window*. Within the frame window's client area is an invisible window, the *MDI client window*, which provides behind-the-scenes management of the dynamically-created child windows called *MDI child windows*.

Figure 14.1
The components of a sample MDI application



The frame window also has a menu, often labeled Window, from which the user can select operations that control the MDI child windows such as Tile, Cascade, Arrange, and Close All. This menu is called the *child window menu*. An entry for each opened MDI child window is automatically appended to the end of this menu. The currently selected window is marked with a check-mark.

Each MDI child window has some characteristics of an overlapped window. It can be maximized to the full size of the frame window or minimized to an icon within the frame window. MDI child windows never appear outside the borders of their frame window. An MDI child window cannot have a menu, so all of its functions are controlled by the frame window's menu. The caption of each MDI child window is often the name of the open file associated with that window, although this behavior is optional and under program control. You can think of an MDI application as a mini-Windows session, complete with many applications represented by windows or icons.

Each MDI window is an object

ObjectWindows defines classes whose instances represent MDI frame and client windows. They are **TMDIFrame** and **TMDIClient**, respectively. The MDI child windows are instances of a class that you derive from **TWindow**.

In an ObjectWindows MDI application, the frame window owns its MDI client window and stores a reference to it in its *ClientWnd* data member. The frame window also holds each of its MDI child windows in its child list.

TMDIFrame's member functions are concerned mainly with construction and management of MDI child windows and the MDI client window, and with processing menu selections. **TMDIClient**'s primary role is behind-the-scenes management of MDI child windows. In designing MDI applications, you will generally derive new classes from **TMDIFrame** and **TWindow** for your frame and child windows, respectively. You'll normally use an instance of **TMDIClient** for your MDI Client window.

Constructing MDI windows

There are a few special considerations for constructing the windows of an MDI application. For a complete description of constructing windows, see Chapter 10, "Window objects."

Constructing MDI frame windows

The MDI application's frame window is also its main window, so it is constructed from within the **InitMainWindow** member function of the application object. However, unlike **TWindow**, **TMDIFrame**'s constructor does not take a parent window (frame windows, being main windows, have no parent), and it takes one more parameter, a resource ID of the menu. MDI frame windows are required to have menus, so **TMDIFrame**'s constructor requires a menu argument and sets *Attr.Menu* for you.

The frame window's menu must include an MDI-style child window menu. This is the menu to which the MDI child window's menu entries will be appended. **TMDIFrame**'s *ChildMenuPos* data member contains the position of the child window member. **TMDIFrame**'s constructor initially sets *ChildMenuPos* to zero, indicating the leftmost top-level menu item. However, you can reset *ChildMenuPos* in your MDI Window's constructor:

```
TMyMDIWindow::TMyMDIWindow(LPSTR ATitle, HMENU AMenu) :
                           TMDIFrame(ATitle, AMenu)
{
  ChildMenuPos = 1;
}
```

TMDIFrame::SetupWindow calls **InitClientWindow** to construct the **TMDIClient** object to serve as its MDI client window. **TMDIFrame::SetupWindow** creates the MDI client window.

Constructing MDI child windows

TMDIFrame defines an automatic response member function called **CMCreateChild** that is invoked when a menu item with a CM_CREATECHILD ID is selected. Usually this menu item is called New or Create. As it is defined by **TMDIFrame**, **CMCreateChild** constructs an MDI child window object by calling **TMDIFrame::CreateChild**. **CreateChild** then calls **InitChild** and the **MakeWindow** member function of the application object to create the window.

TMDIFrame::InitChild just creates a window of type **TWindow** with no caption, so you'll want to redefine **InitChild** for your MDI window class to construct an instance of your MDI child window class:

```
PTWindowsObject TMyMDIWindow::InitChild()
{
  return(new TMyMDIChild(this, "New Child Window"));
}
```

Then if **CreateChild** is called, an instance of your MDI child window class will be constructed and created. You may want your frame window to produce *one* MDI child window when it first appears. Unlike other child windows, MDI child windows must be constructed and created from within the MDI frame window's **SetupWindow** member function, rather than from within the constructor. You will also have to explicitly create the child window:

```
void TMyMDIWindow::SetupWindow()
{
  TMDIFrame::SetupWindow();
  CreateChild();
}
```

If you would like this first MDI child to be maximized, construct the MDI child window object either directly or by calling **InitChild** and add WS_MAXIMIZED to the styles previously set for the object. Then call the **MakeWindow** method of the application object to create the MDI child window:

```
void TMyMDIWindow::SetupWindow()
{
  PMyMDIChild NewChild;

  TMDIFrame::SetupWindow();
  NewChild = new TMyMDIChild(this, "New Child Window");
  NewChild->Attr.Style |= WS_MAXIMIZE;
  GetApplication()->MakeWindow(NewChild);
}
```

In some applications, you will want to create MDI child windows in response to more than one menu choice. For example, New and Open menu choices in a file editor might both bring up a new child window with the file name as caption. In that case, you'll define a command-based response member function for each of the menu items, which will construct and create a child window.

Message processing in an MDI application

As with regular (non MDI) parent and child windows, Windows command-based messages first come to the child window for a chance to intercept and process it. Then the messages go to the parent window. This way, the frame window's menu can be used to control activity in the currently active MDI child window. The frame window also has a chance to respond to selections from its menu.

Managing MDI child windows

The ObjectWindows MDI window classes provide member functions for manipulating the MDI child windows of an MDI application. While much of the underlying work is done by **TMDIClient**, all of the functionality and data is accessible through **TMDIFrame** member functions.

Child window activation

The user of an MDI application is free to activate any open or minimized MDI child window. However, you might want to take some action when the user deactivates one child window by activating another. For example, the frame window's menus might reflect the current state of the active child window through

graying or checking. Whenever a child window becomes active or inactive, it receives the Windows message WM_MDIACTIVATE. You can redefine **TMDIClient::WMMDIActivate** (which normally just calls **DefWndProc**) for the child window object, to respond accordingly.

Child window menu

TMDIFrame defines Windows message response member functions that automatically respond to the standard MDI menu selections: Tile, Cascade, Arrange Icons, and Close All. These member functions expect command-based messages with predefined menu ID constants. Be sure to use these IDs when you build a child window menu resource:

Table 14.1 Standard MDI actions, commands, and member functions

Action	Menu ID constant	TMDIFrame member function
Tile	CM_TILECHILDREN	**CMTileChildren**
Cascade	CM_CASCADECHILDREN	**CMCascadeChildren**
Arrange Icons	CM_ARRANGEICONS	**CMArrangeIcons**
Close All	CM_CLOSECHILDREN	**CMCloseChildren**

TMDIFrame's response member functions, such as **CMTileChildren**, call other **TMDIFrame** member functions, such as **TileChildren**. These member functions call **TMDIClient** member functions of the same name, such as **TMDIClient::TileChildren**. Normally, you won't redefine any of this behavior, but if you want to, you can redefine **TMDIFrame::TileChildren** or the other **TMDIFrame** member functions (not the automatic response methods).

Sample MDI application

The program *MDITest* produces the MDI-compliant application pictured in Figure 14.1.

The complete file, MDITEST.CPP, can be found in the EXAMPLES subdirectory.

C H A P T E R

15

Streamable objects

This chapter describes how the ObjectWindows stream manager provides *persistence* for ObjectWindows objects. The various objects that you create during an ObjectWindows application—windows, dialog boxes, collections, and so on—flower but briefly. They are constructed, displayed, accessed, and destroyed as the application proceeds. Objects can appear and disappear as they enter and leave their scope, then vanish completely when the program terminates. The stream manager lets you save these objects either in memory or file streams so that they *persist* beyond their normal lifespan.

There are countless applications for persistent objects. When saved in shared memory, for example, they can provide interprocess communication. They can be transmitted via modems to other systems. And, most significantly, objects can be saved permanently on disk using file streams. They can then be read back and restored by the same application, by other instances of the same application, or by other applications. Efficient and safe streamability is available to all ObjectWindows objects. All the ObjectWindows classes, including **TButton**, **TDialog**, and **TWindow**, are designed as streamable classes, so streaming them is as easy as normal file I/O.

Building your own streamable classes is also straightforward. Making a class streamable incurs very little overhead. A streamable class needs three virtual functions (**read**, **write**, and **streamableName**), a builder, and a build constructor. The **read**, **write**, and **streamableName** functions are declared as virtual

functions in **TStreamable**. All streamable classes must be derived, directly or indirectly, from **TStreamable**. **TWindowsObject** already has **TStreamable** as one of its bases, so every class derived from **TWindowsObject** is streamable.

The stream objects are simply created using **pstream** and its derived classes. These classes, provided specially for persistent stream I/O, are quite similar to the standard C++ iostream classes, so if you are familiar with them, you have little more to master. For those new to streams, the following section presents a brief introduction and establishes some essential terminology.

The iostream library

Neither C nor C++ has keywords or predefined operators to handle I/O operations. Both rely on functions built into standard libraries: **stdio** in C and **iostream** in C++. The C++ **iostream** library is more flexible, extensible, and secure by providing overloaded operators and typesafe I/O for both standard and user-defined data types. Although the **stdio** functions such as **printf** are available in C++, most C++ programmers prefer the advantages of the stream approach. But what, exactly, is a stream?

A stream is an abstract data type representing an unlimited (in theory, of course) sequence of items with various access properties. Streams have *length* (the number of elements), *current position* (the unique access point at any given moment), and *access mode* (read-only for input, write-only for output, or read/write for combined input/output). Reading (or *extracting*) takes place at the current position (which then usually advances to the next item), but writing (or inserting) is performed by appending items at the end of the stream.

The traditional disk file is one familiar implementation of a stream, but the concept has been extended to cover streams in memory, streams of characters from a keyboard (such as the standard input, **cin**) and to a screen (such as the standard outputs, **cout** and **cerr**), and streams to and from serial ports and other devices. Much of this philosophy originated with UNIX, where "everything is a file." In fact, with the aid of OOP technology, any *source* (or *producer*) of data can be provided with *extraction* methods and treated as an input stream. Similarly most *sinks* (or *consumers*) of data can be given *insertion* methods and treated as output streams. Streams associated with disk files will typically

provide both input and output. This implements a rich hierarchy of stream classes to handle the many variants available. Stream classes can represent various mixes of the following: buffered or unbuffered, formatted or unformatted, in-memory or on file, and each with input, output, and combined input/output versions. ObjectWindows for C++ builds on this edifice to provide stream I/O for the more complex objects used in ObjectWindows.

The overloaded << and >> operators

One of the keys to the success of C++ stream I/O is the convenience of overloaded, chainable operators: **<<** for output (insertion) and **>>** for input (extraction). The complex syntax **printf** and other **stdio** library functions is replaced by simple, elegant statements such as

```
cout << "Hello, World" << endl;
```

Provided that **<<** and **>>** are suitably overloaded for a given data type and stream class, objects of that type can be written to and read from the appropriate stream objects. Such overloading is "built into" C++ for the standard types, such as **char**, **short**, **int**, **long**, **char ***, **float**, **double**, and **void ***. Overloading can be readily extended to user-defined, non-class data types. Achieving the same overloading for class objects is a little more difficult, but it has been done for you in ObjectWindows. You can therefore write and read objects with statements such as:

```
os << AWindow << ADialog;
is >> AButton;
```

without worrying unduly about the inner details. There are a few straightforward, preliminary chores for you when creating and *registering* your own streamable classes; these will be covered shortly. In the previous fragment, *os* is an **opstream** (or derived) object and *is* is an **ipstream** (or derived) object. These two classes, each derived from **pstream**, provide base stream classes for persistent objects (hence the "p" in their names). They are analogous to the **istream** and **ostream** classes, derived from **ios**, in the standard C++ class hierarchy. There is also an **iopstream** class combining **ipstream** and **opstream**. You'll also meet file stream variants called **ifpstream**, **ofpstream**, and **iofpstream**. These correspond to the standard C++ **ifstream**, **ofstream**, and **fstream**

classes and have similar fields and methods. The **pstream** class hierarchy follows.

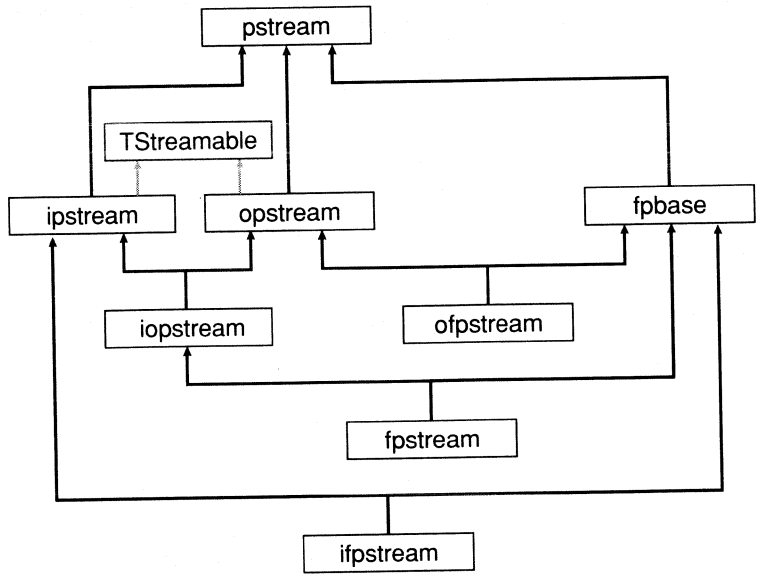


Figure 15.1
Streamable class hierarchy used by ObjectWindows

Overloaded **<<** operators, for example those used for writing **TWindow** objects and object pointers to an **opstream**, are defined in <window.h> as follows:

```
inline Ropstream operator << (Ropstream os, RTWindow cl);
   { return os << (RTStreamable) cl; }
inline Ropstream operator << (Ropstream os, PTWindow cl);
   { return os << (PTStreamable) cl; }
```

In the first variant, the **TWindow** object *cl* will be written to the **opstream** object, *os*. The same output stream is returned by the operator, allowing the familiar chaining of **<<** operations used in the standard C++ **ostream** class. The second variant does the same for a pointer to a **TWindow** object. Note that the operators are implemented by simply typecasting the target object and invoking a call to the **<<** operator defined in a base class. In this case, **TWindow** has a base class **TStreamable**.

Similarly overloaded **<<** and **>>** operators are provided for all the standard streamable classes in ObjectWindows. They are usually implemented as inline, and follow the above pattern. You will usually need to write similar code to overload **<<** and **>>** for your own streamable classes, which are discussed in the following section.

Streamable classes and TStreamable

A streamable class is one whose objects can be written to and read from persistent streams, using the tools provided by the ObjectWindows stream manager. In addition to having suitable I/O operators (usually overloaded **<<** and **>>**, but you can define your own), a streamable class must have **TStreamable** as a base class somewhere in its hierarchy. A glance at the ObjectWindows class tree reveals that **TWindowsObject** is multiply-derived from **Object** and **TStreamable**. All classes derived from **TWindowsObject** therefore inherit the magic of **TStreamable**. All the classes derived from **TWindowsObject** also have overloaded **<<** and **>>** operators, and are therefore streamable. **TScroller** is also multiply-inherited from **TStreamable**, and is therefore also streamable.

The stream manager

A certain amount of background housekeeping is needed when complex objects are saved on and retrieved from streams. This is the job of the stream manager. When data objects are written to a stream, they can be stored and retrieved as straightforward binary images without too many complications. A class object, however, can hold any number of different data types, including pointers to other complex objects, and possibly pointers to the *vtable* (virtual functions table).

To cope with this and other imponderables, the stream manager maintains a database of streamable classes using the **TStreamableTypes**, **TStreamable**, and **TStreamableClass** classes. The overhead is about 12 bytes per streamable class. These classes combine to provide the basic registration, read, write, and build functions for particular objects. The reason for registering a class with the stream manager, in fact, is to ensure that it is known to the stream manager as a streamable class, with a proper entry in this database. The build operation is explained in the next section.

In addition, during stream writes and reads, the stream manager has to maintain a database of all objects written to and read from a stream. Without delving too deeply (since the operation is performed quietly in the background), consider the problem of writing objects to a stream via pointers. Suppose *ptr1* and *ptr2*

point to the same object and you write both **ptr1* and **ptr2* to a stream. When these objects are subsequently read from the stream, you could end up with two identical copies of the object with different pointers, and potential chaos. The stream manager's database of written objects avoids this problem: only one copy of **ptr1* is written to the stream. Similarly, when reading the object from a stream to both **ptr1* and **ptr2*, only only object will be created and *ptr1* and *ptr2* will both point to it.

The stream manager also works closely with the stream objects by providing the correct **read** and **write** functions. **TStreamable** has pure virtual **read** and **write** functions (known as readers and writers) that must be redefined in each streamable class derived from it:

```
class TStreamable {
⋮
protected:
  virtual Pvoid read(Ripstream) = 0;
  virtual void write(Ropstream) = 0;
};
```

The job of the writer is to write all the necessary fields of the object to the stream. Each streamable class must have a writer, but its implementation can be greatly simplified by invoking the writer of its base class. Here is the definition of **TWindow::write**:

```
void TWindow::write(Ropstream os)
{
  long SaveStyle;
  BOOL NameIsNumeric;
  TWindowsObject::write(os);
  if ( !IsFlagSet(WB_FROMRESOURCE) )
  {
    SaveStyle = Attr.Style & ~(WS_MINIMIZE | WS_MAXIMIZE);
    if ( HWindow )
      if ( IsIconic(HWindow) )
        SaveStyle |= WS_MINIMIZE;
      else
        if ( IsZoomed(HWindow) )
          SaveStyle |= WS_MAXIMIZE;
    os << SaveStyle << Attr.ExStyle << Attr.X
       << Attr.Y << Attr.W << Attr.H
       << (long)(Attr.Param);
  }
  os << Attr.Id;

  NameIsNumeric = HIWORD(Attr.Menu) == NULL;
```

```
    os << NameIsNumeric;
    if ( NameIsNumeric )
      os << (long)(Attr.Menu);
    else
      os.fwriteString(Attr.Menu);

    os << Scroller;
  }
```

The read function, called a *reader*, parallels the action of the writer. Each streamable class must redefine the pure virtual read function in **TStreamable**. This is often done by extending its base class's reader to cover any additional fields.

A special case arises in certain reading situations. If you read an object from a stream into an existing object of the appropriate type, the read simply transfers the appropriate data and vtable pointers. However, if there is no such object to be read into, one must be constructed first, a sort of skeleton object ready to be initialized from the stream. This is the role of the *builder*. The builder calls a special **StreamableInit** or *build* constructor that allocates raw memory of sufficient size to hold an object, and constructs its vtable. Each streamable class that might be involved in such reading situations must have a builder and a *build* constructor.

Readers, writers, builders, and build constructors are predefined for the standard ObjectWindows streamable classes. If you want to create your own streamable classes, you need to provide them.

Streamable class constructors

As explained previously, a streamable class whose objects may be the target of a stream read operation needs a special constructor. This constructor must take a single argument *streamableInit*, an **enum** constant defined in objstrm.h. You'll find that all the standard streamable classes have such a constructor, which must be **public**. The pattern is as follows:

```
class TMyStreamable : public TBase, public TStreamable {
⋮
public:
  TMyStreamable(StreamableInit s);
⋮
};
```

The data type **StreamableInit** is an **enum** with the single member *streamableInit*. When you create an object using this constructor:

```
TMyStreamble str(streamableInit);
```

the constructors for any contained pointers are not invoked, as would be the case with standard constructors. The result is simpler memory management. When you read an object from a stream to the object *str*, the reader for the object doesn't have to free any unwanted data from *str*:

```
TMyStreamable str(streamableInit);
// create "empty" str
⋮
ifpstream ifps("str.sav");      // open file stream for i/p
ifps >> str;                    // read object to str
⋮
```

One word of caution: objects created with the *streamableInit* constructor, such as *str*, cannot be safely used *until* they have been "initialized" by a stream-read operation. Incidentally, the enum and member names have no special significance: they simply provide a unique data type argument to distinguish this constructor from any others.

The **build** member function for a streamable class is defined as follows:

```
PTStreamable TMyStreamableClass::build()
{
  return new TMyStreamableClass(streamableInit);
}

TMyStreamableClass::TMyStreamableClass(StreamableInit s) :
  TBaseClass(streamableInit)
{
}
```

In other words, the build constructor simply calls that of its base, and so on down the line.

Streamable class names

Each streamable class needs to override the private virtual function, **streamableName**, inherited from **TStreamable**. This function must return a unique name for the class as a null-terminated string:

```
class TMyStreamable : public TBase, public TStreamable {
:
private:
  virtual const Pchar streamableName() const
  { return "TMyStreamable"; }
};
```

Again, this chore has already been done for all the standard ObjectWindows streamable classes. But if you define your own streamable classes, you must provide your own overrides. The normal approach is to have **streamableName** return the class name as shown above. The stream manager uses this unique name in the various class databases it maintains.

Using the stream manager

There are three steps to using the stream manager:

- Link in the stream manager code
- Create a suitable stream object
- Use the stream object

Let's examine each step in detail.

Linking in the stream manager code

To handle its objects, every streamable class must define the following three functions: **read**, **write**, and **build**. These functions must be known to the stream manager and linked into any application that uses the stream manager. This linkage, which is known as *registering* the class name with the stream manager, is achieved by using the __link macro, whose purpose is to provide a reference to an object of type **TStreamableClass**. By convention, the name of this object is **Reg*classname***, excluding the initial *T* of the class name, if there is one. For example, if you have a class **TBase** defined in BASE.CPP as follows

```
class TBase : public TStreamable
{
  int x;
  int y;
};
```

and a class **TDerived** defined in MYAPP.CPP that contains a data member that is a pointer to a **TBase** object, you must use the

__link macro to inform the stream manager to link in stream support for **TBase**. Stream class registration and the definition for **TDerived** follows:

```
__link(RegBase)
class TDerived : public TStreamable
{
  int z;

  PTBase zz;
};
```

Remember that the P-types are pointers.

Use the __link macro whenever you have a data member that is a pointer to a class that's defined outside of the source file.

Creating a stream object

Creating a **ipstream** or **opstream** stream object requires a declaration with suitable arguments, exactly as with the **iostream** classes. To save a dialog on a file stream called DLG.SAV, you just declare an **ofpstream** object as follows:

For the various stream constructors and their arguments, see Chapter 17, "Streamable class reference".

```
TChDirDialog cdlg;
⋮
// create the dialog here

ofpstream of("dlg.sav");
// open an output file stream

of << cdlg;
// write the dialog object to the file stream
```

Here, the constructor

```
ofpstream(PCchar filename, int mode = ios::out,
          int prot = filebuf::openprot)
```

has been invoked with the defaults indicated.

Using the stream object

A typical read operation might be this:

```
TChDirDialog cdlg (streamableInit);
// invoke the build constructor; create skeleton object

ifpstream ifps("dlg.sav");
// open an input file stream

ifps >> cdlg;
// read the dialog object from the file stream to cdlg
```

Collections on streams

C++ container classes are not streamable by default. The following section provides an example of how to derive a new container class from an existing one to make it streamable. The discussion also provides step-by-step instructions for making a class streamable and, in the process, explains how each required component of the implementation is used by the stream manager. The following example is STEP9.CPP, found in the EXAMPLES\STEPS directory.

Making Array streamable

The program uses the class **Array** (from the container class library) to store a collection of **TPoint**s (an object defined in this sample program). Class **Array** is not by itself streamable, so a new class, **TPointArray**, will be defined. **TPointArray** inherits multiply from both **Array** and **TStreamable**. All streamable classes must inherit from **TStreamable**.

The streamable builder function

All streamable classes must define a **build** static member function, a special constructor, and three virtual functions: **read**, **write**, and **streamableName**. The following class definition provides the declarations for each of these member functions:

```
class TPointArray : public Array, public TStreamable {

public:
  TPointArray(int upper, int lower = 0, sizeType aDelta = 0) :
    Array(upper, lower, aDelta) {};
⋮
public:
  static PTStreamable build();
protected:
  TPointArray(StreamableInit) : Array(50, 0, 50) {};
  virtual void write(Ropstream);
  virtual Pvoid read(Ripstream);

private:
  const Pchar streamableName() const {
    return "TPointArray"; }
};
```

This code fragment writes a **TPointArray** to a stream:

```
ofpstream os("save.dat");
PTPointArray APointArray;
⋮
os << APointArray;
```

Although in many cases, it's not strictly necessary, each streamable class should define *inserters* and *extractors* that perform a type conversion between the object and the **TStreamable** component of the object, as shown here:

```
inline Ripstream operator >> (Ripstream is, RTPointArray cl)
  { return is >> (RTStreamable)cl; }

inline Ripstream operator >> (Ripstream is, RPTPointArray cl)
  { return is >> (RPvoid)cl; }

inline Ropstream operator << (Ropstream os, RTPointArray cl)
  { return os << (RTStreamable)cl; }

inline Ropstream operator << (Ropstream os, PTPointArray cl)
  { return os << (PTStreamable)cl; }
```

In this example code fragment, a pointer to a **TPointArray** is written to the stream, *os*. Also, the fourth inline definition is invoked, causing a call to the inserter (*operator* <<) whose argument is a **PTStreamable**. This inserter is defined by the stream implementation and results in a call to the **write** virtual member function of the object supplied as a parameter. In the example, **TPointArray::write** is invoked.

```
void TPointArray::write(Ropstream os) {
  RContainerIterator PointIterator = initIterator();
  os << getItemsInContainer();

  while (int(PointIterator) != 0) {
    RObject PointObject = PointIterator++;
    if (PointObject != NOOBJECT)
    {
      os << ((PTPoint)(&PointObject))->X;
      os << ((PTPoint)(&PointObject))->Y;
    }
  }

  delete &PointIterator;
}
```

TPointArray::write allocates an iterator for the **Array** container, writes to the stream the number of elements in the array, then iterates through the array, writing out each element of the array, and finally destroying the iterator object. A somewhat more modular approach would have been to make the class **TPoint**

streamable; that way, the loop could simply have written the **TPoint** object to the stream rather than writing out each of **TPoint**'s data members (*X* and *Y*).

The streamableName member function

When an object is written to a stream, the stream manager invokes the **streamableName** virtual member function of the object to determine its name. This name is written to the stream prior to writing out the object. The definition of the inline **streamableName** member function follows, from the class definition.

```
class TPointArray : public Array, public TStreamable {
⋮
private:
  const Pchar streamableName() const
    { return "TPointArray"; }
⋮
};
```

When an object is read from a stream, the stream manager first reads in the string representation of the class name. This name has been registered with the stream manager through a call to the static **TStreamableClass** constructor. All streamable classes must create a static instance of type **TStreamableClass** to perform this registration:

```
TStreamableClass RegPointArray("TPointArray", TPointArray::build,
                               __DELTA(TPointArray));
```

This constructor specifies the name of the class ("TPointArray"), the address of the builder function (**TPointArray::build**), and the offset from the base of the object to the TStreamable component of the object. The _ _DELTA macro automatically calculates this offset.

The stream manager maintains a database of all the streamable classes through the use of these static objects. When an object of type **TPointArray** is read in from a stream, it is looked up in this database to find the address of the builder static member function, **TPointArray::build**.

```
PTStreamable TPointArray::build()
{
  return new TPointArray(streamableInit);
}
```

All builder functions simply invoke a special constructor, whose argument is of type **StreamableInit**. This constructor invokes its base class constructors, but does not initialize any of the data members. Thus, the created object is the correct size and has its virtual table filled in.

The definition for the constructor can be found above, in the class definition. The type **StreamableInit** is an **enum** with one value: *streamableInit*. The special constructor for streaming is selected by the **enum** parameter from all the possible constructors for a class.

Streamable reader function

With a valid object now available, the stream manager can invoke the **read** virtual member function. Its definition is provided here:

```
Pvoid TPointArray::read(Ripstream is) {
  sizeType NumPoints;
  PTPoint APoint;

  is >> NumPoints;

  for (int i = 0; i < NumPoints; ++i)
  {
    APoint = new TPoint(0, 0);
    is >> APoint->X;
    is >> APoint->Y;
    add(* (APoint));
  };

  return this;
}
```

In the example, the **TPointArray** reader function reads in the number of elements in the array, and then reads in the X,Y value pairs associated with each **TPoint**. It then adds the new **TPoint** to the **Array**.

As mentioned earlier, a better approach would have been to make **TPoint** itself streamable. If this had been done, and you had implemented the **TPoint** streamability in a different file, you would have had to tell the linker to link in the streamable support for the **TPoint** object. Assuming that the **TStreamableClass** registration object had been called **RegPoint**, you could have forced the streamable support to be linked in through the use of the __link macro as shown here:

```
__link(RegPoint)
```

This would cause an external reference to the object **RegPoint**, of type **TStreamableClass**. Since this registration object contains a pointer to the static member builder function, it causes that function to be linked in. It, in turn, causes the special streamable constructor and the virtual reader and writer functions to get linked in.

PART 3

ObjectWindows Reference

C H A P T E R

16

Class reference

This chapter alphabetically lists all the standard ObjectWindows classes. It explains their use, members, operators, and friends. Figure 16.1 shows the ObjectWindows hierarchy.

How to find it

To find information on a specific class, keep in mind that many of the properties of the classes in the hierarchy are inherited from base classes. Rather than endlessly duplicate that information, this chapter only documents data members and member functions that are *new* or *redefined* for a particular class.

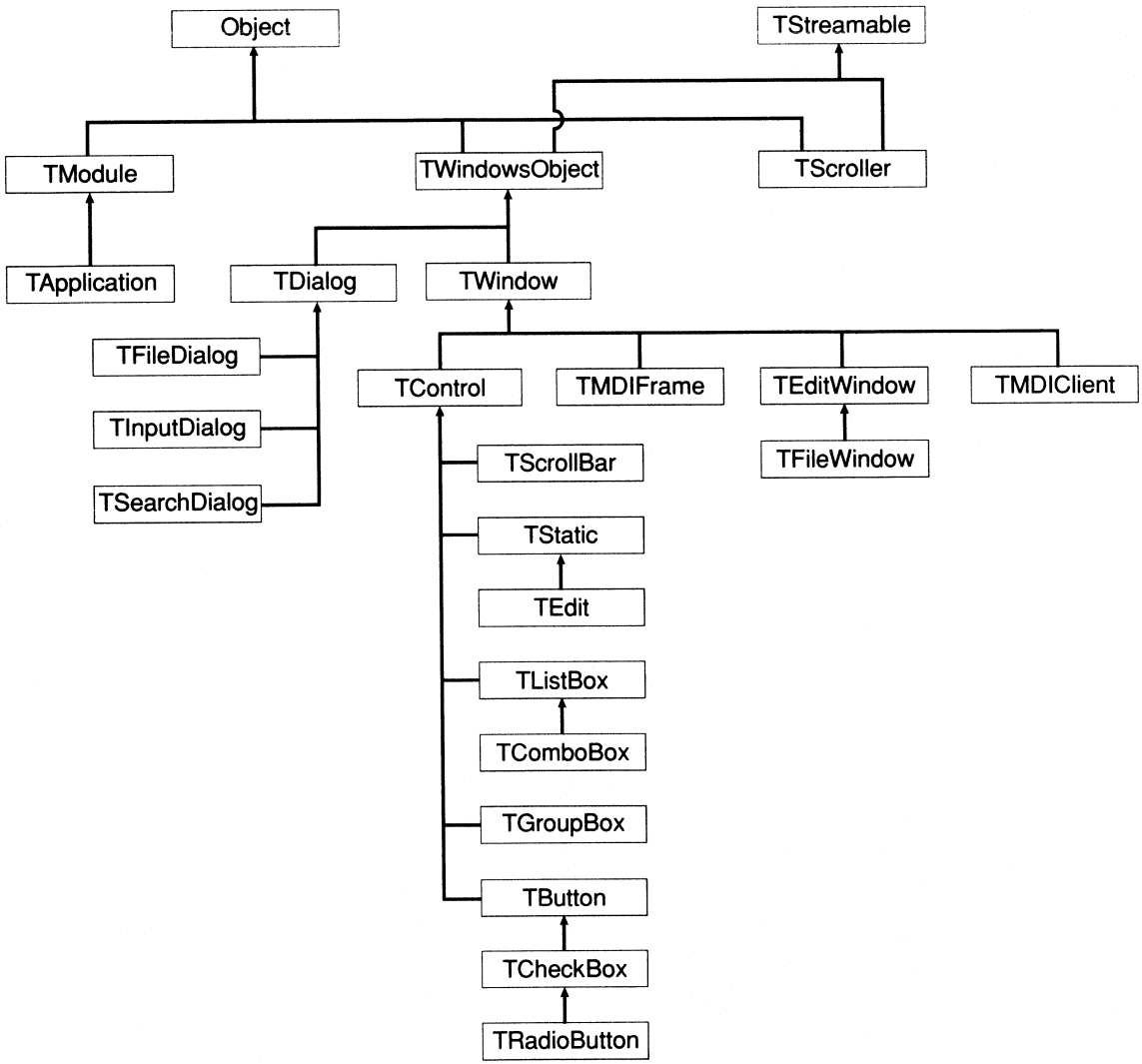
Use the index to find information for a particular inherited member. However, you can also search back up the hierarchy. The diagrams at the top of each class entry in this book show the ancestry of that class, except the classes **Object** and **TStreamable**. For example, the class **TControl** is inherited from **TWindow**, which is inherited from **TWindowsObject**, which is inherited from **Object** and **TStreamable**. So the diagram for **TControl** shows **TWindowsObject**, **TWindow**, and **TControl**. All members of each class are listed.

For example, if you want to know about the *Parent* data member of a **TEdit** class, you would start at the right of the diagram, first looking under **TEdit**, where you won't find *Parent* listed. You would then check the class to the left of **TEdit** in the diagram, **TStatic**. Again, *Parent* will not be listed, so you'll check in the class to the left of **TStatic**, and so on to the left, until you get to **TWindowsObject**, where you find an entry for *Parent*. Then you

can turn to the entry for **TWindowsObject** (using the index or thumbtabs) and read about *Parent*.

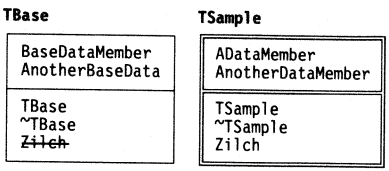
When a member is ~~struck out~~, that means it is redefined in a later class.

Figure 16.1: ObjectWindows class hierarchy



The entry for each class is laid out in the following format:

TSampleClassName class — sample class header file



This section provides a general overview of the class, relationship with other classes, and general use. It is followed by listings of data members, member functions, and related functions (if any).

For more information on ObjectWindows types or constants (such as EM_OUTOFMEMORY), see Chapter 18, "Miscellaneous components." For more information on Windows types (such as LPSTR), see the online Help.

All functions with **protected** member access are labeled to the right of the program text (declarations). All functions with **public** member access are not labeled. Private members are not listed.

Data members

This section lists all data members that are newly introduced for each class, alphabetically. Each member's declaration, member access (if protected), and an explanation of use is provided.

ADataMember

```
SomeType ADataMember;
```

ADataMember is a data member that holds some information about this sample class. This text explains what it contains, and how you use it.

See also: Related data members, member functions, classes, miscellaneous functions, and so on.

AnotherDataMember

```
AnotherType AnotherDataMember;
```

protected

AnotherDataMember has similar information to that for *ADataMember*.

Member functions

This section lists all member functions that either are newly defined for this class or redefine inherited member functions.

constructor `TSampleClassName(SomeType AParameter);`

Constructor for a new sample class; sets the *ADataMember* data member to *AParameter*.

Zilch `virtual void Zilch();`

The **Zilch** member function causes the sample class to perform some action.

See also: **TSomethingElse::Zilch**

Object — object.h

Object is the root of the ObjectWindows hierarchy. As an abstract class, **Object** defines what a derived object must do. It provides the basic mechanism and structure for type checking and encapsulation.

Data members

ZERO `static PObject ZERO;`

Pointer to an error object used by a private class. It handles problems when the operator **new** cannot allocate space for an object.

Member functions

constructor `Object();`

Constructs an instance of **Object**.

constructor `Object(RObject);`

Copy constructor (copies data members onto another instance of **Object**).

destructor `virtual ~Object();`

Virtual destructor (destroys an instance of **Object**).

firstThat `virtual RCObject firstThat(condFuncType, Pvoid) const;`

Calls the specified test function for an object with a parameter list. Finds the first object in the container that satisfies a given condition. Returns the object if the test succeeded; NOOBJECT otherwise.

forEach

```
virtual void forEach(iterFuncType, Pvoid);
```

Calls the given iterator for an object, along with a parameter list. Iterates through each object in the container.

hashValue

```
virtual hashValueType hashValue() const = 0;
```

Pure virtual function for derived classes; returns a hash value, a unique key based on the object.

isA

```
virtual classType isA() const = 0;
```

Pure virtual function for derived classes; returns a unique class identifier for the class.

isAssociation

```
virtual int isAssociation() const;
```

Determines if the object is derived from the class **Association**. Returns 0 for the base **Object** class.

isEqual

```
virtual int isEqual(RCObject) const = 0;
```

Pure virtual function for derived classes; tests for equality.

isSortable

```
virtual int isSortable() const;
```

Determines if the object can be sorted; that is, if it is derived from **Sortable**. Returns 0 for the base **Object** because it is not sortable.

lastThat

```
virtual RCObject lastThat(condFuncType, Pvoid) const;
```

Same as **FirstThat** for non-container objects. Redefined for container objects to return a reference to the last object for which the given conditional function returns a 1.

nameOf

```
virtual Pchar nameOf() const = 0;
```

Pure virtual function for derived classes; returns a class identification string.

new

```
Pvoid operator new(size_t);
```

Allocates a given number of bytes for an object. If the allocation fails, returns *ZERO*.

printOn

```
virtual void printOn(Rostream) const = 0;
```

Pure virtual function for derived classes; writes a printable representation of the object on a stream.

Friend

operator <<

```
Rostream operator <<(Rostream, RCObject);
```

Object friend function that writes an object value on an output stream.

Related functions

operator ==

```
int operator ==(RCObject test1, RCObject test2);
```

Determines whether the first object is equal to the second object.

operator !=

```
int operator !=(RCObject test1, RCObject test2);
```

Determines whether the first object is not equal to the second object.

Operators >> and <<

Streamable classes all declare four operators as related functions. The classes include **TButton**, **TCheckBox**, **TComboBox**, **TDialog**, **TEdit**, **TEditWindow**, **TFileDialog**, **TFileWindow**, **TInputDialog**, **TGroupBox**, **TListBox**, **TMDIClient**, **TMDIFrame**, **TRadioButton**, **TScrollBar**, **TScroller**, **TStatic**, **TWindow**, and **TWindowsObject**. We say ***TClassName*** to refer to any one of these classes.

There are two each of **operator >>** and **operator <<**. These operators are frequently unnecessary, but will ensure no ambiguities arise in cases of multiple inheritance.

The two **operator >>** functions differ in that the first takes *a reference* to a ***TClassName*** object and the second takes *a reference to a pointer* to a ***TClassName*** object. Likewise, the two**operator <<** functions differ in that the first takes *a reference* to a ***TClassName*** object and the second takes *a pointer* to a ***TClassName*** object.

operator >>

```
Ripstream operator >> (Ripstream is, RTClassName cl);
Ripstream operator >> (Ripstream is, RPTClassName cl);
```

Reads a ***TClassName*** object from the input stream *is* and writes it to *cl*. A reference to the stream is returned, permitting the usual chaining of **>>** operators.

See also: **ipstream**

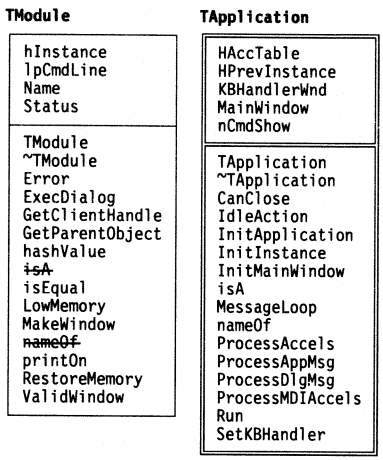
operator <<

```
Ropstream operator << (Ropstream os, RTClassName cl);
Ropstream operator << (Ropstream os, PTClassName cl);
```

Writes the ***TClassName*** object *cl* to the output stream *os*. A reference to the stream is returned, permitting the usual chaining of **<<** operators.

See also: **opstream**

TApplication — applicat.h



TApplication is derived from **TModule** and acts as an object-oriented stand-in for a Windows application module. **TApplication** and **TModule** supply the basic behavior required of a Windows application. **TApplication** member functions perform instance initialization, including the creation of a main window, and message processing.

Data members

HAccTable

```
HANDLE HAccTable;
```

HAccTable holds a handle to the current Windows accelerator table being used by the application.

hPrevInstance

```
HANDLE hPrevInstance;
```

Contains the handle of the previously executing instance of the Windows application. If 0, there were no previously executing instances when this instance began execution.

KBHandlerWnd

```
PTWindowsObject KBHandlerWnd;
```

KBHandlerWnd points to the currently active window if that window's keyboard handler mechanism is enabled. ObjectWindows uses this member to allow a window with controls to process keyboard input, shifting the focus from control to control when tabs or arrow keys are pressed.

MainWindow

```
PTWindowsObject MainWindow;
```

MainWindow points to the application's main window.

nCmdShow

```
int nCmdShow;
```

An integer that indicates how the main window is to be displayed (normally, or as an icon). See the Windows function **ShowWindow** for the values. ObjectWindows uses it to display the main window of the application.

Member functions

constructor

```
TApplication(LPSTR AName, HANDLE AnInstance, HANDLE APrevInstance,
             LPSTR ACmdLine, int ACmdShow);
```

Constructs a **TApplication** object, initializing its data members using the specified parameters. Invokes the **TModule** constructor.

destructor

```
~TApplication();
```

Destroys the **TApplication** object. If the **MainWindow** still exists, deletes it.

CanClose

```
virtual BOOL CanClose();
```

Returns TRUE if it's OK for the application to close. By default, it calls the **CanClose** member function of its main window and returns its return value. This member function is seldom redefined; closing behavior can be redefined in the main window's **CanClose** member function.

See also: **TWindowsObject::CanClose, TWindowsObject::WMDestroy**

IdleAction

```
virtual void IdleAction();                                    protected
```

Invoked during the message loop when the application is idle. Performs processing when there are no messages in the Windows message queue for any application. Is a "do-nothing" method which can be redefined in derived classes to specify processing during this "idle" time.

InitApplication `virtual void InitApplication();` **protected**

Performs any initialization necessary only for the first executing instance of the application. By default **InitApplication** does nothing. Derived classes can redefine **InitApplication** to perform application-specific initialization.

See also: **TApplication::InitInstance, TApplication::Run**

InitInstance `virtual void InitInstance();` **protected**

Performs any initialization necessary for every executing instance of the application. **InitInstance** calls **InitMainWindow**, and creates and shows the main window element by calling **TModule::MakeWindow** and **TWindowsObject::Show**. If the main window could not be created, the *Status* data member is set to EM_INVALIDMAINWINDOW. If you redefine this member function, be sure to explicitly call **TApplication::InitInstance**.

See also: **TApplication::InitMainWindow, TApplication::InitApplication, TApplication::Run, TModule::MakeWindow, TWindowsObject::Show**

InitMainWindow `virtual void InitMainWindow();` **protected**

By default, **InitMainWindow** constructs a generic **TWindow** object with the application name as its caption. Redefine **InitMainWindow** to construct a useful main window object of a derived class and store it in *MainWindow*. Typical use:

```
virtual void TMyApp::InitMainWindow(){
   MainWindow = new TMyWindow(NULL, "Caption");
}
```

isA `virtual classType isA() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *applicationClass*, the class identifier of **TApplication**.

MessageLoop `virtual void MessageLoop();` **protected**

Operates the application's message loop, which runs during the lifetime of the application. Queries Windows for messages and if one is received, processes it by calling **ProcessAppMsg**. If the query returns without a message, **MessageLoop** calls **IdleAction** to perform some processing during the idle time.

See also: **TApplication::ProcessAppMsg, TApplication::IdleAction**

nameOf `virtual Pchar nameOf() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns "TApplication", the class identification string of **TApplication**.

ProcessAccels `virtual BOOL ProcessAccels(LPMSG PMessage);` **protected**

Handles special accelerator message processing. If your application's windows do not respond to accelerators, you can improve performance by overriding this member function to simply return FALSE.

See also: **TApplication::ProcessAppMsg**

ProcessAppMsg `virtual BOOL ProcessAppMsg(LPMSG PMessage);` **protected**

Checks for special processing for modeless dialog box, accelerator, and MDI accelerator messages. Calls **ProcessDlgMsg**, **ProcessMDIAccels**, and **ProcessAccels** and returns TRUE if any of these special messages are encountered. If your application does not create modeless dialog boxes, does not respond to accelerators, and is not an MDI application, you can improve performance by overriding this member function to simply return FALSE.

See also: **TApplication::ProcessDlgMsg**, **TApplication::ProcessAccels**, **TApplication::ProcessMDIAccels**

ProcessDlgMsg `virtual BOOL ProcessDlgMsg(LPMSG PMessage);` **protected**

Handles special modeless dialog box and window message processing for handling keyboard input for controls. If your application creates no modeless dialog boxes or windows with controls, you can improve performance by overriding this member function to simply return FALSE.

See also: **TApplication::ProcessAppMsg**

ProcessMDIAccels `virtual BOOL ProcessMDIAccels(LPMSG PMessage);` **protected**

Handles special accelerator message processing for MDI-compliant applications. If your application is not an MDI application, you can improve performance by overriding this member function to simply return FALSE.

See also: **TApplication::ProcessAppMsg**

Run `virtual void Run();`

Initializes the instance, calling **InitApplication** if no other instances are already in execution; then calls **InitInstance**. Sets the application in motion

by calling **MessageLoop** if initialization was successful (that is, if the *Status* data member is 0).

See also: **TApplication::InitApplication, TApplication::InitInstance, TApplication::MessageLoop**

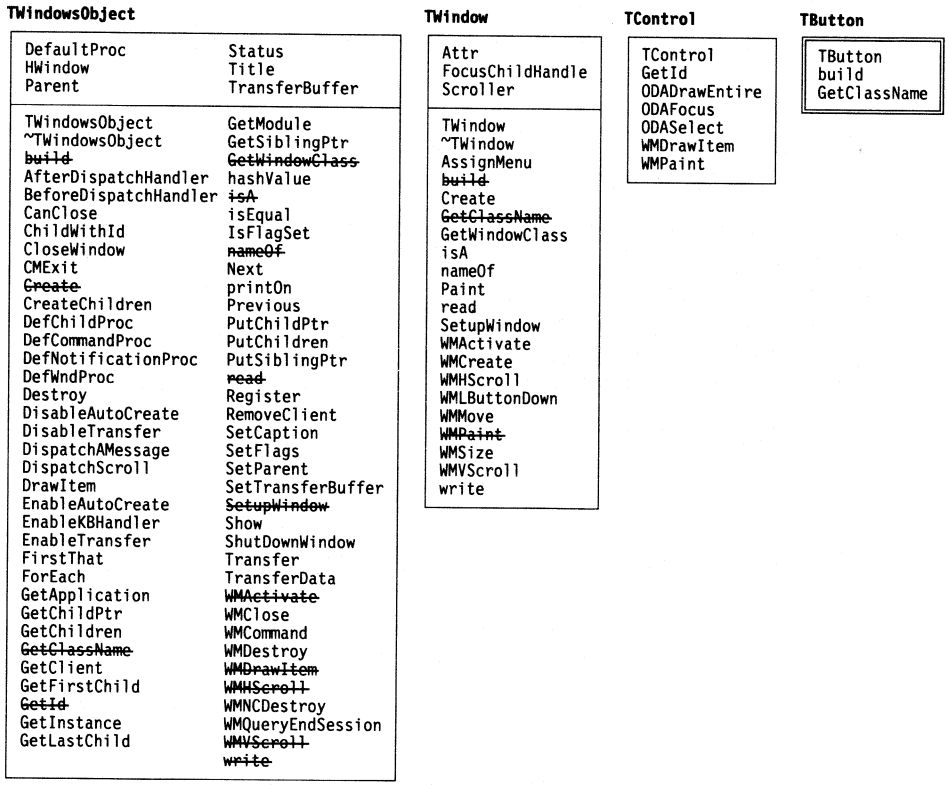
SetKBHandler

```
void SetKBHandler(PTWindowsObject AWindowsObject);
```

Activates keyboard handling (translation of keyboard input into control selections) for the given window by setting *KBHandlerWnd* to *AWindowsObject*. (Called internally by ObjectWindows.)

See also: **TWindowsObject::EnableKBHandler, TWindowsObject::WMActivate**

TButton — button.h



TButton is an interface class that represents a push button interface element in Windows. You must use a **TButton** to create a button control in

a parent **TWindow**. You can also use a **TButton** to facilitate communication between your application and the button controls of a **TDialog**.

There are two types of push buttons. A regular button appears with a thin border. A default button appears with a thick border and represents the default action of the window (if the user presses *Enter*). There can only be one default push button in a window.

Data Member

IsDefPB

```
BOOL IsDefPB;
```

Indicates whether the button is to be considered the default push button. Used for owner-draw buttons. Set by a **TButton** constructor based on BS_DEFPUSHBUTTON style setting.

Member functions

constructor

```
TButton(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y,
        int W, int H, BOOL IsDefault, PTModule AModule = NULL);
```

Constructs a button object with the supplied parent window (*AParent*), control ID (*AnId*), associated text (*AText*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), and height (*H*). Invokes **TControl**'s constructor with similar parameters. Adds BS_DEFPUSHBUTTON to the default styles set for the **TButton** (in *Attr.Style*), if *IsDefault* is TRUE. Otherwise, it adds BS_PUSHBUTTON.

See also: **TControl::TControl**

constructor

```
TButton(PTWindowsObject AParent, int ResourceId,
        PTModule AModule = NULL);
```

Constructs a **TButton** object to be associated with a button control of a **TDialog**. Invokes the **TControl** constructor with identical parameters. Then calls **DisableTransfer** to exclude the button from the transfer mechanism because they have no data to be transferred.

The *ResourceId* parameter must correspond to a button resource that you define.

See also: **TControl::TControl, TWindowsObject::DisableTransfer**

constructor `TButton(StreamableInit);` **protected**

TButton stream constructor. Invokes **TControl**'s stream constructor. Called when a **TButton** is instantiated using data from an input stream.

See also: **TControl::TControl**

build `static PTStreamable build();`

Invokes the **TButton(StreamableInit)** constructor. Constructs an object of the type **TButton** prior to reading in its data members from a stream.

BMSetStyle `virtual void BMSetStyle(RTMessage Msg) = [WM_FIRST + BM_SETSTYLE];`

For owner-draw buttons keeps track of whether the owner-draw button is the default push button when Windows tries to set the style to BS_DEFPUSHBUTTON. Otherwise just calls **DefWndProc**.

GetClassName `virtual LPSTR GetClassName();` **protected**

Returns the name of **TButton**'s Windows registration class, "BUTTON".

See also: **TWindowsObject::GetWindowClass**

SetupWindow `virtual void SetupWindow();` protected

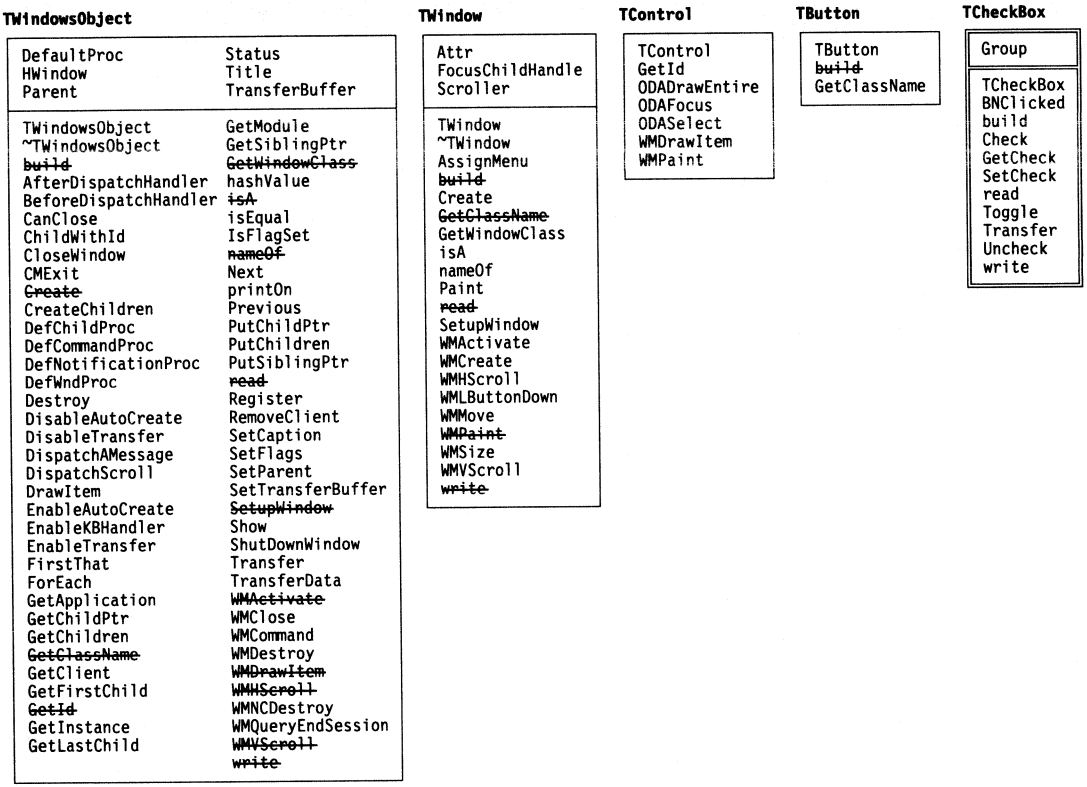
Sends a DM_SETDEFID message to the parent window if the button is the default push button and an owner-draw button.

WMGetDlgCode `virtual void WMGetDlgCode(RTMessage Msg) = [WM_FIRST + WM_GETDLGCODE];`

Responds to WM_GETDLGCODE messages from the dialog manager. For owner-draw buttons returns information as to whether the button is the default button. Otherwise it just calls DefWndProc.

TCheckBox

checkbox.h



TCheckBox is an interface object that represents a check box interface element in Windows. You must use a **TCheckBox** to create a check box control in a parent **TWindow**. You can use a **TCheckBox** to facilitate communication between your application and the check box controls of a **TDialog**.

Check boxes have two check states: *checked* and *unchecked*. Three-state check boxes have an additional *grayed* state. **TCheckBox** member functions are concerned primarily with managing the check box's state. Optionally, a check box can be part of a group (**TGroupBox**) that visually and functionally groups its controls.

Data member

Group

```
PTGroupBox Group
```

Group points to the **TGroupBox** control object that groups the check box logically with other check boxes and radio buttons (**TRadioButton**). If the check box is not part of a group, *Group* is equal to NULL.

See also: **TGroupBox**, **TRadioButton**

Member functions

constructor

```
TCheckBox(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y,
          int W, int H, PTGroupBox AGroup, PTModule AModule = NULL);
```

Constructs a check box object with the specified parent window (*AParent*), control ID (*AnId*), associated text (*ATitle*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), and associated group box (*AGroup*). **TCheckBox::TCheckBox** sets the check box's *Attr.Style* data member to WS_CHILD | WS_VISIBLE | WS_TABSTOP | BS_AUTOCHECKBOX. Invokes a **TButton** constructor.

See also: **TButton::TButton**

constructor

```
TCheckBox(PTWindowsObject AParent, int ResourceId, PTGroupBox AGroup,
          PTModule AModule = NULL);
```

Constructs a **TCheckBox** object to be associated with a check box control of a **TDialog**. *ResourceId* must correspond to a check box resource in the resource file. Invokes a **TButton** constructor. Sets *Group* to *AGroup*. Then enables the data transfer mechanism by calling **EnableTransfer**.

See also: **TButton::TButton**, **TWindowsObject::EnableTransfer**

constructor

```
TCheckBox(StreamableInit);
```

protected

TCheckBox stream constructor. Invokes **TButton**'s stream constructor. Called when a **TCheckBox** is instantiated using data from an input stream.

See also: **TButton::TButton**

BNClicked

```
virtual void BNClicked(RTMessage Msg)
  = [NF_FIRST + BN_CLICKED];
```

protected

Automatically responds to notification messages indicating the check box was clicked. If the check box's *Group* is not NULL, **BNClicked** notifies the

TGroupBox that its check state has changed by calling its **SelectionChanged** member function.

See also: **TGroupBox::SelectionChanged**

build

```
static PTStreamable build();
```

Invokes the **TCheckBox(StreamableInit)** constructor. Constructs an object of the type **TCheckBox** prior to reading in its data members from a stream.

Check

```
void Check();
```

Forces the check box into the checked state by calling **SetCheck**.

See also: **TCheckBox::SetCheck**

GetCheck

```
WORD GetCheck();
```

Returns BF_UNCHECKED if the check box is unchecked, BF_CHECKED if it is checked, or BF_GRAYED if it is grayed.

SetCheck

```
void SetCheck(WORD CheckFlag);
```

Forces the check box into the check state specified by *CheckFlag*. The box will be checked, unchecked, or grayed depending upon the *CheckFlag* supplied (BF_CHECKED, BF_UNCHECKED, or BF_GRAYED). If the check box is part of a group, **SetCheck** informs the group that the selection has changed.

See also: **TGroupBox::SelectionChanged**

read

```
virtual Pvoid read(Ripstream is);
```

protected

Invokes **TWindow::read** to read in the base **TWindow** object. Then reads in *Group* using **GetSiblingPtr**.

Toggle

```
void Toggle();
```

Toggles the check state of the check box. For two-state check boxes, toggles between checked and unchecked states. For three-state check boxes, toggles between checked, unchecked, and grayed states.

See also: **TCheckBox::SetCheck**

Transfer

```
virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);
```

Transfers the check state of the check box as a WORD-type value (BF_CHECKED, BF_UNCHECKED, or BF_GRAYED) to or from a transfer buffer pointed to by *DataPtr*. If *TransferFlag* is TF_GETDATA, the check box's state data is transferred to the buffer. If *TransferFlag* is TF_SETDATA, the check box is set to the state contained in the transfer buffer. **Transfer**

returns the size of the transfer data (in bytes). To retrieve the size of the data without transfer, supply TF_SIZEDATA as the *TransferFlag*.

Uncheck `void Uncheck();`

Forces the check box into the unchecked state by calling **SetCheck**.

See also: **TCheckBox::SetCheck**

write `virtual void write(Ropstream os);` **protected**

Invokes **TWindow::write** to write out the base **TWindow** object. Then writes out *Group* using **PutSiblingPtr**.

TComboBox — combobox.h

TComboBox is an interface object that represents a combo box interface element in Windows. A **TComboBox** must be used to create a combo box control in a parent **TWindow**. A **TComboBox** may also be used to facilitate

communication between your application and the combo box controls of a **TDialog**. **TComboBox** objects inherit most of their behavior from **TListBox**.

There are three types of combo boxes: *simple*, *drop down*, and *drop down list*. These types are governed by the style constants CBS_SIMPLE, CBS_DROPDOWN, and CBS_DROPDOWNLIST. These constants are supplied to the constructor of a **TComboBox**, which tells Windows which type of combo box element to create.

Data member

TextLen

```
WORD TextLen;
```

TextLen-1 is the maximum length of the character buffer in the edit portion of the combo box. *TextLen* is set by the constructor of **TComboBox**.

Member functions

constructor

```
TComboBox(PTWindowsObject AParent, int AnId, int X, int Y, int W, int H,
          DWORD AStyle, WORD ATextLen, PTModule AModule = NULL);
```

Constructs a combo box object with the specified parent window (*AParent*), control ID (*AnId*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), style (*AStyle*), and text length (*ATextLen*).

Invokes the **TListBox** constructor with similar parameters. Then sets *Attr.Style* as follows:

```
Attr.Style = WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP | CBS_SORT |
             CBS_AUTOHSCROLL | WS_VSCROLL | AStyle;
```

One of the following combo box style constants must be among the styles set in *AStyle*: CBS_SIMPLE, CBS_DROPDOWN, CBS_DROPDOWNLIST, CBS_OWNERDRAWFIXED, or CBS_OWNERDRAWVARIABLE.

See also: TComboBox::TextLen, **TListBox::TListBox**

constructor

```
TComboBox(PTWindowsObject AParent, int ResourceId, WORD ATextLen,
          PTModule AModule = NULL);
```

Constructs a **TComboBox** object to be associated with a combo box control of a **TDialog**. Invokes the **TListBox** constructor that has similar parameters, then sets the *TextLen* member to *ATextLen*.

The *ResourceId* parameter must correspond to a combo box resource that you define.

See also: **TListBox::TListBox**

constructor

```
TComboBox(StreamableInit);
```

protected

TComboBox stream constructor. Invokes **TListBox**'s stream constructor. Called when a **TComboBox** is instantiated using data from an input stream.

See also: **TListBox::TListBox**

build

```
static PTStreamable build();
```

Invokes the **TComboBox(StreamableInit)** constructor. Constructs an object of the type **TComboBox** prior to reading in its data members from a stream.

Clear

```
void Clear();
```

Clears the text of the associated edit control.

GetClassName

```
virtual LPSTR GetClassName();
```

protected

Returns the name of **TComboBox's** Windows registration class, "COMBOBOX".

GetEditSel

```
int GetEditSel(Rint StartPos, Rint EndPos);
```

Returns, in the specified reference parameters, the starting and ending positions of the text selected in the associated edit control. Returns CB_ERR if the combo box has no edit control.

GetMsgID

```
virtual WORD GetMsgID(WORD AnId);
```

protected

Returns the identifier of the Windows combo box message associated with the supplied ObjectWindows message identifier. Allows instances of **TComboBox** to inherit many **TListBox** member functions.

GetText

```
int GetText(LPSTR AString, int MaxChars);
```

Fills the specified *AString* with the text of the associated edit control (up to *MaxChars*). Returns the number of characters copied.

GetTextLen `int GetTextLen();`

Returns the length of the text in the associated edit control.

HideList `void HideList();`

Hides the list of a drop down or drop down list combo box.

See also: **TComboBox::ShowList**

nameOf `virtual Pchar nameOf()const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns "TComboBox", the class identification string of **TComboBox**.

read `virtual Pvoid read(Ripstream is);` **protected**

Invokes **TListBox::read** to read in the base **TListBox** object. Then reads in *TextLen*.

SetEditSel `int SetEditSel(int StartPos, int EndPos);`

Selects characters which are between *StartPos* and *EndPos* in the edit control of the combo box. Returns CB_ERR if the combo box does not have an edit control.

SetText `void SetText(LPSTR AString);`

Selects the first string in the associated list box which begins with the supplied AString. If there is no match, sets the contents of the associated edit control to the supplied string (and selects it).

SetupWindow `virtual void SetupWindow();` **protected**

Sets up the combo box by limiting the length of the text in its edit control to *TextLen* – 1.

ShowList `void ShowList();`

Shows the list of a drop down or drop down list combo box.

See also: **TComboBox::HideList**

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

Transfers the items and selection of the combo box to or from a transfer buffer if TF_SETDATA or TF_GETDATA, respectively, is passed as the *TransferFlag*. *TransferBuffer* is expected to point to a **PTComboBoxData** object which is used as a data transfer buffer for a combo box.

Transfer returns the size of **PTComboBoxData** (the pointer, not the structure). To retrieve the size without transferring data, pass TF_SIZEDATA as the *TransferFlag*.

You must use a pointer in your transfer buffer to these structures. You cannot imbed copies of the structures in your transfer buffer. And you can't use these structures as transfer buffers.

See also: **TListBox::Transfer**

write `virtual void write(Ropstream os);` **protected**

Invokes **TListBox::write** to write out the base **TListBox** object. Then writes out *TextLen*.

TControl — control.h

TControl unifies derived control classes, such as **TScrollBar** and **TButton**. Control objects of derived classes are used to represent control interface elements in Windows. A control object must be used to create a control in a parent **TWindow**. A control object can be used to facilitate communication between your application and the controls of a **TDialog**.

Member functions

constructor

```
TControl(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y,
        int W, int H, PTModule AModule = NULL);
```

Invokes **TWindow**'s constructor, passing it *AParent* (parent window), *ATitle* (caption text), and *AModule*. Sets *Attr* using the supplied window identifier (*AnId*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), and height (*H*) parameters. It sets *Attr.Style* to WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP.

See also: **TWindow::TWindow**

constructor

```
TControl(PTWindowsObject AParent, int ResourceId,
        PTModule AModule = NULL);
```

Constructs an object to be associated with an interface control of a **TDialog**. Invokes the **TWindow** constructor, then enables the data transfer mechanism by calling **EnableTransfer**.

The *ResourceId* parameter must correspond to a control interface resource that you define.

See also: **TWindow::TWindow, TWindowsObject::EnableTransfer**

constructor

```
TControl(StreamableInit);
```

protected

TControl stream constructor. Invokes **TWindow**'s stream constructor. Called when a **TControl** is instantiated using data from an input stream.

See also: **TWindow::TWindow**

GetId

```
virtual int GetId();
```

Returns the window identifier, *Attr.Id*.

ODADrawEntire

```
virtual void ODADrawEntire(DRAWITEMSTRUCT _FAR & DrawInfo);
```

protected

Responds to a notification sent to a drawable control when the control needs to be drawn. By default, calls **Parent->DrawItem**. Can be redefined by a drawable control to specify the manner in which it is to be drawn.

See also: **TWindowsObject::DrawItem, TControl::WMDrawItem**

ODAFocus `virtual void ODAFocus(DRAWITEMSTRUCT _FAR & DrawInfo);` **protected**

Responds to a notification sent to a drawable control when the focus has shifted to or from the control. By default, calls **Parent->DrawItem**. Can be redefined by a drawable control to specify the manner in which it is to be drawn when losing or gaining the focus.

See also: **TWindowsObject::DrawItem, TWindowsObject::WMDrawItem**

ODASelect `virtual void ODASelect(DRAWITEMSTRUCT _FAR & DrawInfo);` **protected**

Responds to a notification sent to a drawable control when the selection state of the control changes. By default, calls **Parent->DrawItem**. Can be redefined by a drawable control to specify the manner in which it is drawn when its selection state changes.

See also: **TWindow::DrawItem, TWindow::WMDrawItem**

WMDrawItem `virtual void WMDrawItem(RTMessage Msg) = [WM_FIRST + WM_DRAWITEM];`

Responds to a WM_DRAWITEM message sent to a drawable control when the control needs to be drawn. **TControl::WMDrawItem** calls **ODADrawEntire** if the entire control needs to be drawn. **ODASelect** is called if the selection state of the control has changed. **ODAFocus** is called if the focus has been shifted to or from the control.

See also: **TControl::ODADrawEntire, TControl::ODASelect, TControl::ODAFocus, TWindowsObject::WMDrawItem**

WMPaint `virtual void WMPaint(RTMessage Msg) = [WM_FIRST + WM_PAINT];`

If the control has a predefined Windows class, calls **DefWndProc** for Windows-supplied painting. Otherwise, calls **TWindow::WMPaint**.

See also: **TWindow::WMPaint**

TDialog

dialog.h

TDialog objects represent both modal and modeless dialog box interface elements. A **TDialog** has a corresponding resource definition that describes the placement and appearance of its controls. The identifier of this resource definition is supplied to the constructor of the **TDialog** object.

A **TDialog** is associated with a modal or modeless interface element by calling its **Execute** or **Create** member function, respectively. These member functions, however, are not normally called directly in an ObjectWindows application. Instead, **TModule**'s **ExecDialog** and **MakeWindow** member functions are called, which check the validity of a **TDialog** prior to its association.

A modal dialog box disables operations in its parent window while it is open.

Data members

Attr

```
TDialogAttr Attr;
```

Attr holds the dialog creation attributes ofthe dialog box. **TDialogAttr** is defined in Chapter 18, "Miscellaneous components."

IsModal

```
BOOL IsModal;
```

IsModal is TRUE if the dialog box is modal and FALSE if it is modeless.

Member functions

constructor

```
TDialog(PTWindowsObject AParent, LPSTR AName, PTModule AModule = NULL);
```

Invokes a **TWindowsObject** constructor, passing *AParent* and *AModule*. Calls **DisableAutoCreate** so that the **TDialog** will not be automatically created and displayed along with its parent. Intitializes *Title* to –1. Assigns a copy of *AName* to *Attr.Name*. *AName* is a string resource identifier that you must define in the resource definition file. Initializes *Attr.Param* to 0. Sets *IsModal* to FALSE.

See also: **TWindowsObject::TWindowsObject, TWindowsObject::DisableAutoCreate**

constructor

```
TDialog(PTWindowsObject AParent, int ResourceId,
        PTModule AModule = NULL);
```

Invokes a **TWindowsObject** constructor, passing *AParent* and *AModule*. Calls **DisableAutoCreate** so that the **TDialog** will not be automatically created and displayed along with its parent. Intitializes *Title* to –1. Sets *Attr.Name* using the dialog's integer resource identifier. It must correspond to a dialog resource definition in the resource file. Initializes *Attr.Param* to 0. Sets *IsModal* to FALSE.

See also: **TWindowsObject::TWindowsObject, TWindowsObject::DisableAutoCreate**

constructor

```
TDialog(StreamableInit);
```

protected

TDialog stream constructor. Invokes **TWindowsObject**'s stream constructor. Called when a **TDialog** is instantiated using data from an input stream.

See also: **TWindowsObject::TWindowsObject**

destructor `virtual ~TDialog();`

If *Attr.Name* is a string and not an integer resource identifier, frees memory allocated to hold the name of the **TDialog** (*Attr.Name*).

See also: **TWindowsObject::~TWindowsObject**

build `static PTStreamable build();`

Invokes the **TDialog(StreamableInit)** constructor. Constructs an object of the type **TDialog** prior to reading in its data members from a stream.

Cancel `virtual void Cancel(RTMessage Msg) = [ID_FIRST + IDCANCEL];` **protected**

Automatic response to a click on the Cancel button of the dialog box. Calls **CloseWindow(int)** passing IDCANCEL.

See also: **TDialog::CloseWindow**

CloseWindow `virtual void CloseWindow(int ARetValue);`

Conditionally shuts down the dialog box. If **this** is a modeless dialog box calls **TWindowsObject::CloseWindow**. If **this** is a modal dialog box, calls **CanClose**. If **CanClose** returns TRUE, calls **TransferData** to transfer dialog box data then calls **ShutDownWindow**, passing *ARetValue*.

See also: **TWindowsObject::CloseWindow, TDialog::CanClose, TDialog::TransferData, TDialog::ShutDownWindow**

CloseWindow `virtual void CloseWindow();`

Conditionally shuts down the dialog box. Calls **TWindowsObject::CloseWindow** if **this** is a modeless dialog box. If **this** is a modal dialog box, calls **CloseWindow** passing IDCANCEL as the integer *ARetValue*.

See also: **TWindowsObject::CloseWindow**

Create `virtual BOOL Create();`

Creates a modeless dialog box interface element associated with the **TDialog** object. (The association is not attempted if *Status* is nonzero.) Calls **DisableAutoCreate** to prevent automatic creation of all child windows. Calls **EnableKBHandler** to enable keyboard handling. Registers all the dialog's child windows for custom control support. Calls the Windows function **CreateDialogParam** to create the dialog box.

Create returns TRUE if successful. If unsuccessful, *Status* is set to EM_INVALIDWINDOW and **Create** returns FALSE.

☞ **TModule::MakeWindow** is a safer way to create a modeless dialog box.

See also: **TDialog::Execute, TModule::MakeWindow, TWindowsObject::DisableAutoCreate, TWindowsObject::EnableKBHandler**

Destroy `virtual void Destroy(int ARetValue);`

Destroys the interface element associated with the **TDialog**. If **this** is a modeless dialog box, calls **TWindowsObject::Destroy**. If **this** is a modal dialog box, calls **EnableAutoCreate** on all child windows. Then calls the Windows function **EndDialog**, passing *ARetValue* for return as the result of the dialog's execution.

See also: **TWindowsObject::Destroy, TWindowsObject::EnableAutoCreate**

Destroy `virtual void Destroy();`

Destroys the interface element associated with the **TDialog**. If **this** is a modeless dialog box, calls **TWindowsObject::Destroy**. If **this** is a modal dialog box, calls **Destroy**, passing IDCANCEL as the integer *ARetValue*.

See also: **TWindowsObject::Destroy**

Execute `virtual int Execute();`

Creates and executes a modal dialog box interface element associated with the **TDialog** object. The association is not attempted if the *Status* member is nonzero. If successfully asociated, does not return until the **TDialog** is closed.

Calls **DisableAutoCreate** to prevent automatic creation of all child windows. Calls **EnableKBHandler** to enable keyboard handling. Registers all the dialog's child windows for custom control support. Calls the Windows function **DialogBoxParam** to execute the dialog box.

☞ **TModule::ExecDialog** is a safer way to create a modal dialog box.

See also: **TModule::ExecDialog, TWindowsObject::DisableAutoCreate, TWindowsObject::EnableKBHandler**

GetClassName `virtual LPSTR GetClassName();` **protected**

Returns the name of the dialog box's default Windows class, which must be used for a modal dialog box. Returns the name of the default ObjectWindows class ("OWLDialog") for a modeless dialog box.

GetItemHandle

```
HWND GetItemHandle(int DlgItemID);
```

Returns the dialog box control's window handle identified by the supplied ID, *DlgItemID*.

GetWindowClass

```
virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);
```

protected

Fills *AWndClass* with the default registration attributes for a **TDialog**. By default, a **TDialog** has a standard application icon and arrow cursor. To change these or other default registration attributes, redefine this member function (and also redefine **GetClassName**) in your **TDialog** derived class. Be sure to call **TDialog::GetWindowClass** in your **GetWindowClass** member function prior to modifying the defaults which **TDialog** sets.

See also: **TWindowsObject::GetClassName**

isA

```
virtual classType isA() const;
```

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *dialogClass*, the class identifier of **TDialog**.

nameOf

```
virtual Pchar nameOf()const;
```

Required by container classes. Redefines the pure virtual function in class **Object**. Returns "TDialog", the class identification string of **TDialog**.

Ok

```
virtual void Ok(RTMessage Msg) = [ID_FIRST + IDOK];
```

protected

Responds to a click on the dialog box's OK button (with the identifier IDOK). Calls **CloseWindow** (passing IDOK).

See also: **TDialog::CloseWindow**

read

```
virtual Pvoid read(Ripstream is);
```

protected

Invokes **TWindowsObject::read** to read in the base **TWindowsObject** object. Then reads in *Attr.Name* and *IsModal*.

SendDlgItemMsg

```
DWORD SendDlgItemMsg(int DlgItemID, WORD AMsg, WORD WParam,
                     DWORD LParam);
```

Sends a Windows control message, identified by *AMsg*, to the dialog box's control identified by its supplied ID, *DlgItemID*. *WParam* and *LParam* become parameters in the control message. **SendDlgItemMsg** returns the value returned by the control, or 0 if the control ID is invalid.

SetCaption

```
void SetCaption(LPSTR ATitle);
```

If *ATitle* is not –1, calls **TWindowsObject::SetCaption**.

See also: **TDialog::SetupWindow, TWindowsObject::SetCaption**

SetupWindow `virtual void SetupWindow();` **protected**

Sets up the dialog box by calling **SetCaption** (sets *Title*) and **TWindowsObject::SetupWindow**.

See also: **TDialog::SetCaption, TWindowsObject::SetupWindow**

ShutDownWindow `virtual void ShutDownWindow();`

Shuts down the dialog box. If **this** is a modeless dialog box calls **TWindowsObject::ShutDownWindow**. If **this** is a modal dialog box, calls **Destroy**, passing IDCANCEL.

See also: **TWindowsObject::ShutDownWindow, TDialog::Destroy**

ShutDownWindow `virtual void ShutDownWindow(int ARetValue);`

Shuts down the dialog box. If **this** is a modeless dialog box calls **TWindowsObject::ShutDownWindow**. If **this** is a modal dialog box, calls **Destroy**, passing *ARetValue*.

See also: **TWindowsObject::ShutDownWindow, TDialog::Destroy**

WMClose `virtual void WMClose(RTMessage Msg) = [WM_FIRST + WM_CLOSE];` **protected**

Responds to an incoming WM_CLOSE message by shutting the window down.

See also: **TDialog::ShutDownWindow**

WMInitDialog
```
virtual void WMInitDialog(RTMessage Msg)
  = [WM_FIRST + WM_INITDIALOG];
```
protected

WMInitDialog is automatically called just before the dialog box is displayed. It calls **SetupWindow** to perform any setup required for the dialog box or its controls.

See also: **TWindowsObject::SetupWindow**

MQueryEndSession
```
virtual void WMQueryEndSession(RTMessage Msg)
  = [WM_FIRST + WM_QUERYENDSESSION];
```
protected

Responds to Windows' attempt to close down. If **this** is the main window, calls **GetApplication()->CanClose**; otherwise, calls **this->CanClose**. Returns 0 (sets *Msg.Result*) if the **CanClose** call returns TRUE (indicating that it is OK to shut down). Otherwise, returns 1 (because a dialog box needs to invert the usual test).

This is the opposite of Windows' WM_QUERYENDSESSION behavior.

write `virtual void write(Ropstream os);` **protected**

Invokes **TWindowsObject::write** to write out the base **TWindowsObject** object. Then writes out *Attr.Name* and *IsModal*.

TEdit edit.h

A **TEdit** is an interface object that represents an edit control interface element in Windows. A **TEdit** object must be used to create an edit control in a parent **TWindow**. A **TEdit** can be used to facilitate communication between your application and the edit controls of a **TDialog**.

There are two styles of edit control objects: single line and multiline. Multiline edit controls allow editing of multiple lines of text. Note that the position of the first character in an edit control is 0. For a multiline edit control, position numbers continue sequentially from line to line; line breaks count as two characters.

Most of **TEdit**'s member functions manage the edit control's text. **TEdit** also includes some automatic member response functions that respond to

selections from the edit control's parent window menu, including cut, copy, and paste. Two important member functions inherited from **TEdit's** base class, **TStatic**, are **GetText** and **SetText**.

Member functions

constructor

```
TEdit(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y,
      int W, int H, WORD ATextLen, BOOL Multiline,
      PTModule AModule = NULL);
```

Constructs an edit control object with a parent window (*AParent*). Sets the creation attributes of the edit control and fills its *Attr* data members with the specified control ID (*AnId*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), and height (*H*).

If text buffer length (*ATextLen*) is 0 or 1, there is no explicit limit to the number of characters that can be entered. Otherwise *ATextLen* – 1 characters can be entered. By default, initial text (*AText*) in the edit control is left-justified and the edit control has a border. Multiline edit controls have horizontal and vertical scroll bars.

constructor

```
TEdit(PTWindowsObjectAParent, int ResourceId, WORD ATextLen,
      PTModule AModule = NULL);
```

Constructs a **TEdit** object to be associated with an edit control of a **TDialog**. Invokes the **TStatic** constructor with identical parameters. The *ResourceId* parameter must correspond to an edit resource that you define. Enables the data transfer mechanism by calling **EnableTransfer**.

See also: **TStatic::TStatic**

constructor

```
TEdit(StreamableInit);
```

protected

TEdit stream constructor. Invokes **TStatic**'s stream constructor. Called when a **TEdit** is instantiated using data from an input stream.

See also: **TStatic::TStatic**

build

```
static PTStreamable build();
```

Invokes the **TEdit(StreamableInit)** constructor. Constructs an object of the type **TEdit** prior to reading in its data members from a stream.

CanUndo

```
BOOL CanUndo();
```

Returns TRUE if it is possible to undo the last edit.

See also: **TEdit::Undo**

ClearModify

```
void ClearModify();
```

Resets the change flag of the edit control causing **IsModified** to return FALSE. The flag is set when text is modified.

See also: **TEdit::IsModified**

CMEditClear

```
virtual void CMEditClear(RTMessage Msg)
  = [CM_FIRST + CM_EDITCLEAR];
```

protected

Automatically responds to a menu selection with a menu ID of CM_EDITCLEAR by calling **Clear**.

See also: **TStatic::Clear**

CMEditCopy

```
virtual void CMEditCopy(RTMessage Msg)
  = [CM_FIRST + CM_EDITCOPY];
```

protected

Automatically responds to a menu selection with a menu ID of CM_EDITCOPY by calling **Copy**.

See also: **TEdit::Copy**

CMEditCut

```
virtual void CMEditCut(RTMessage Msg)
  = [CM_FIRST + CM_EDITCUT];
```

protected

Automatically responds to a menu selection with a menu ID of CM_EDITCUT by calling **Cut**.

See also: **TEdit::Cut**

CMEditDelete

```
virtual void CMEditDelete(RTMessage Msg)
  = [CM_FIRST + CM_EDITDELETE];
```

protected

Automatically responds to a menu selection with a menu ID of CM_EDITDELETE by calling **DeleteSelection**.

See also: **TEdit::DeleteSelection**

CMEditPaste

```
virtual void CMEditPaste(RTMessage Msg)
  = [CM_FIRST + CM_EDITPASTE];
```

protected

Automatically responds to a menu selection with a menu ID of CM_EDITPASTE by calling **Paste**.

See also: **TEdit::Paste**

CMEditUndo

```
virtual void CMEditUndo(RTMessage Msg)
  = [CM_FIRST + CM_EDITUNDO];
```

protected

Automatically responds to a menu selection with a menu ID of CM_EDITUNDO by calling **Undo**.

See also: **TEdit::Undo**

Copy

```
void Copy();
```

Copies the currently selected text into the Clipboard.

See also: **TEdit::CMEditCopy**

Cut

```
void Cut();
```

Deletes the currently selected text and copies it into the Clipboard.

See also: **TEdit::CMEditCut**

DeleteLine

```
BOOL DeleteLine(int LineNumber);
```

Deletes the text in the line specified by *LineNumber* in a multiline edit control. If –1 passed, deletes the current line. *DeleteLine* does not delete the line break and affects no other lines. Returns TRUE if successful. Returns FALSE if *LineNumber* is not –1 and is out of range or if an error occurs.

DeleteSelection

```
BOOL DeleteSelection();
```

Deletes the currently selected text, and returns FALSE if no text is selected.

See also: **TEdit::CMEditDelete**

DeleteSubText

```
BOOL DeleteSubText(int StartPos, int EndPos);
```

Deletes the text between the starting and ending positions specified by *StartPos* and *EndPos*, respectively. **DeleteSubText** returns TRUE if successful.

ENErrSpace

```
virtual void ENErrSpace(RTMessage Msg)
   = [NF_FIRST + EN_ERRSPACE];
```

protected

Responds to an error notification message, sent when the edit control unsuccessfully attempts to allocate more memory, by sounding a beep.

GetClassName

```
virtual LPSTR GetClassName();
```

protected

Returns the name of **TEdit**'s Windows registration class, "EDIT".

GetLine

```
BOOL GetLine(LPSTR ATextString, int StrSize, int LineNumber);
```

Retrieves a line of text (whose line number is supplied) from the edit control and returns it in *ATextString* (NULL-terminated). *StrSize* indicates how many characters to retrieve. *GetLine* returns FALSE if it is unable to retrieve the text or if the supplied buffer is too small.

See also: **TStatic::GetText, TEdit::GetNumLines, TEdit::GetLineLength**

GetLineFromPos `int GetLineFromPos(int CharPos);`

From a multiline edit control, returns the line number on which the character position specified by *CharPos* occurs. If *CharPos* is greater than the position of the last character, the number of the last line is returned. If –1 is supplied, the number of the line that contains the first selected character is returned.

GetLineIndex `int GetLineIndex(int LineNumber);`

From a multiline edit control, returns the number of characters that appear before the line number specified by *LineNumber*. If LineNumber is –1, returns the number of the line that contains the caret.

GetLineLength `int GetLineLength(int LineNumber);`

Returns, from a multiline edit control, the number of characters in the line specified by *LineNumber*. If it is –1, the following applies: returns the length of the line where the caret is positioned if no text is selected; if text is selected on the line, returns the line length minus the number of selected characters; if selected text spans more than one line, returns the length of the lines minus the number of selected characters.

GetNumLines `int GetNumLines();`

Returns the number of lines that have been entered in a multiline edit control, 0 if no text or if an error occurs.

GetSelection `void GetSelection(Rint StartPos, Rint EndPos);`

Retrieves the starting and ending positions of the currently selected text and returns them in the *StartPos* and *EndPos* arguments. By using **GetSelection** in conjunction with **GetSubText**, you can get the currently selected text.

See also: **TEdit::GetSubText**

GetSubText `void GetSubText(LPSTR ATextString, int StartPos, int EndPos);`

Retrieves the text in an edit control from indices *StartPos* to *EndPos* and returns it in *ATextString*.

See also: **TEdit::GetSelection**

Insert `void Insert(LPSTR ATextString);`

Inserts the text supplied in *ATextString* into the edit control at the current text insertion point (cursor position), and replaces any currently selected text. **Insert** is similar to **Paste**, but does not affect the Clipboard.

See also: **TEdit::Paste**

IsModified

```
BOOL IsModified();
```

Returns TRUE if the user has changed the text in the edit control.

See also: **TEdit::ClearModify**

Paste

```
void Paste();
```

Inserts text from the Clipboard into the edit control at the current text insertion point (cursor position).

See also: **TEdit::CMEditPaste**

Scroll

```
void Scroll(int HorizontalUnit, int VerticalUnit);
```

Scrolls a multiline edit control horizontally and vertically by the numbers of characters specified in *HorizontalUnit* and *VerticalUnit*. Positive values result in a scroll to the right or down, and negative values scroll to the left or up.

Search

```
int Search(int StartPos, LPSTR AText, BOOL CaseSensitive);
```

Performs a case-sensitive or case-insensitive search for the supplied text. Selects the text if found. Returns the position of the text or –1 if not found. If –1 is supplied as the *StartPos*, then the search starts from the end of the text currently selected, or from the position of the caret if no text is selected.

SetSelection

```
BOOL SetSelection(int StartPos, int EndPos);
```

Forces the selection of the text between the positions specified by *StartPos* and *EndPos*, but not including the character at *EndPos*.

SetupWindow

```
virtual void SetupWindow();
```

protected

If the *TextLen* data member is nonzero, limits the number of characters that can be entered into the edit control to *TextLen* – 1.

See also: **TStatic::TextLen**

Undo

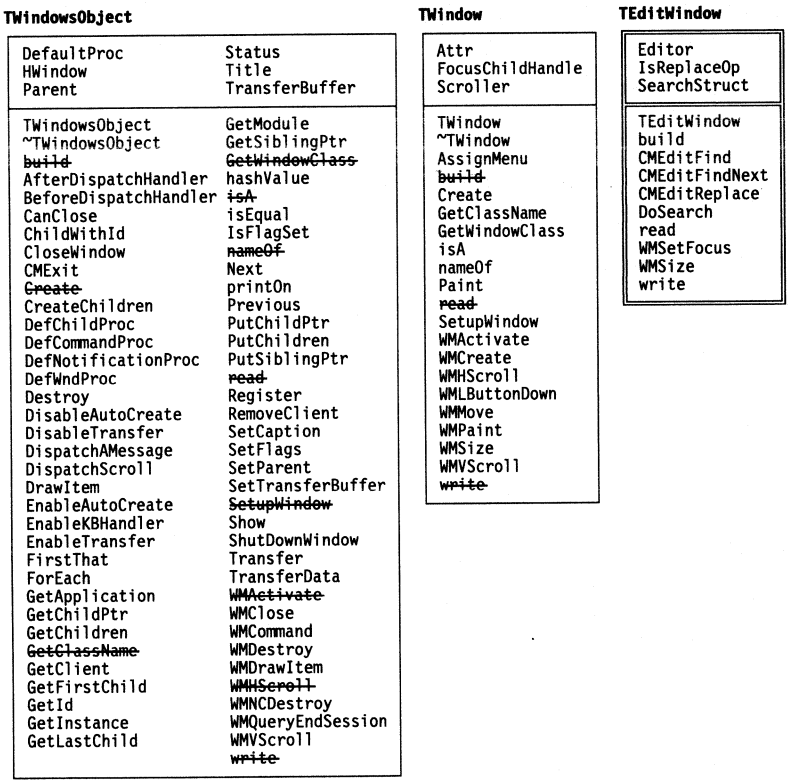
```
void Undo();
```

Undoes the last edit.

See also: **TEdit::CanUndo**, **TEdit::CMEditUndo**

TEditWindow

editwnd.h



TEditWindow defines a specialized window class that allows text entry and editing. It contains a full-window edit control that manages text editing. **TFileWindow,** a specialized descendant of **TEditWindow,** allows editing of files.

Data members

Editor `PTEdit Editor;`

Editor points to a full-window, multiline edit control that provides the edit window with text-editing capabilities.

IsReplaceOp

```
BOOL IsReplaceOp;
```

Contains TRUE if a replace operation will be performed along with the next search operation.

SearchStruct

```
TSearchStruct SearchStruct;
```

A transfer buffer for use with **TSearchDialog**.

See also: **TSearchDialog**

Member functions

constructor

```
TEditWindow(PTWindowsObject AParent, LPSTR ATitle,
            PTModule AModule = NULL);
```

Instantiates the *Editor* edit control, zeroes the *SearchStruct*, and sets *IsReplaceOp* to NULL. *ATitle* specifies the window caption (in the title bar) and is passed to the **TWindow** constructor.

constructor **protected**

```
TEditWindow(StreamableInit);
```

TEditWindow stream constructor. Invokes **TWindow**'s stream constructor. Called when a **TEditWindow** is instantiated using data from an input stream.

build

```
static PTStreamable build();
```

Invokes the **TEditWindow(StreamableInit)** constructor. Constructs an object of the type **TEditWindow** prior to reading in its data members from a stream.

CMEditFind **protected**

```
virtual void CMEditFind(RTMessage Msg)
  = [CM_FIRST + CM_EDITFIND];
```

Responds to the selection of a Find menu item. Executes a search dialog box that retrieves search text and options (transferred into *SearchStruct*) from the user. Sets *IsReplaceOp* to FALSE, then calls **DoSearch**.

See also: **TEditWindow::DoSearch, TSearchStruct**

CMEditFindNext **protected**

```
virtual void CMEditFindNext(RTMessage Msg)
  = [CM_FIRST + CM_EDITFINDNEXT];
```

Responds to the selection of a Find Next menu item by calling **DoSearch**. Assumes that *SearchStruct* already contains text and search/replace options.

See also: **TEditWindow::DoSearch, TSearchStruct**

CMEditReplace

```
virtual void CMEditReplace(RTMessage Msg)
  = [CM_FIRST + CM_EDITREPLACE];
```

protected

Responds to the selection of a Replace menu item by calling **DoSearch**. Executes a search dialog box that retrieves search/replace text and options (transferred into *SearchStruct*) from the user. See **TSearchStruct** (Chapter 18, "Miscellaneous components"). Sets *IsReplaceOp* to TRUE, and calls **DoSearch**.

See also: **TEditWindow::DoSearch**

DoSearch

```
void DoSearch();
```

Searches for text in the *Editor* using text and options in the *SearchStruct*. Displays message boxes that report errors and retrieve confirmation for replacing text, when appropriate. See **TSearchStruct** (Chapter 18, "Miscellaneous components").

See also: **TEdit::Search**

read

```
virtual Pvoid read(Ripstream is);
```

protected

Invokes **TWindow::read** to read in the base **TWindow** object. Then calls **GetChildPtr** to read in the edit control that is a child window.

WMSetFocus

```
virtual void WMSetFocus(RTMessage Msg)
  = [WM_FIRST + WM_SETFOCUS];
```

protected

Responds to an incoming WM_SETFOCUS message by setting the focus to the *Editor* edit control.

WMSize

```
virtual void WMSize(RTMessage Msg) = [WM_FIRST + WM_SIZE];
```

protected

Responds to an incoming WM_SIZE message by sizing the *Editor* edit control to the **TEditWindow**'s client area.

write

```
virtual void write(Ropstream os);
```

protected

Invokes **TWindow::write** to write out the base **TWindow** object. Then calls **PutChildPtr** to write out the edit control child window.

TFileDialog

filedial.h

TFileDialog creates dialog boxes that prompt the user to select a file name when opening or saving a file. Two dialog box resource definitions, for an open and a save file dialog box, are supplied in the dialog box resource FILEDIAL.DLG.

Data members

Extension

```
char Extension[MAXEXT];
```

Contains the file-name extension.

FilePath `LPSTR FilePath;`

Points to the buffer returning the user-supplied file name. When passed to the constructor of the file dialog box, contains the initial file mask.

FileSpec `char FileSpec[FILESPEC];`

Contains the current file name.

PathName `char PathName[MAXPATH];`

Contains the current file path.

Member functions

constructor

```
TFileDialog(PTWindowsObject AParent, int ResourceId, LPSTR AFilePath,
            PTModule AModule = NULL);
```

Invokes **TDialog**'s constructor, passing *AParent*, *ResourceId*, and *AModule*. Sets *FilePath* to *AFilePath*. Uses the contents of the buffer pointed to by *AFilePath* as a template to fill in the file list box. Specify either SD_FILEOPEN (File Open dialog) or SD_FILESAVE (File Save dialog) as the *ResourceId* (integer resource identifier) of the dialog box.

See also: **TDialog::TDialog**

constructor `TFileDialog(StreamableInit);` **protected**

TFileDialog stream constructor. Called when a **TFileDialog** is instantiated using data from an input stream.

build `static PTStreamable build();`

Invokes the **TFileDialog(StreamableInit)** constructor. Constructs an object of the type **TFileDialog** prior to reading in its data members from a stream.

CanClose `virtual BOOL CanClose();`

Returns TRUE if the user entered a valid file name. Retrieves the text of the edit control and updates *PathName*. Calls **UpdateListBoxes**; returns FALSE if **UpdateListBoxes** returns TRUE. Returns FALSE if the edit control contains an invalid file name.

See also: **TFileDialog::UpdateListBoxes**

HandleDList

```
virtual void HandleDList(RTMessage Msg)
  = [ID_FIRST + ID_DLIST];
```

protected

Responds to messages from the directory list box. Updates *PathName* when the selection in the list box changes. Calls **UpdateListBoxes** if the selected entry was double-clicked; otherwise, calls **UpdateFileName**. Clears the selection in the list box when it loses the focus.

See also: **TFileDialog::UpdateListBoxes**

HandleFList

```
virtual void HandleFList(RTMessage Msg)
  = [ID_FIRST + ID_FLIST];
```

protected

Responds to messages from the file list box. Updates *PathName* with the name of the selected file and calls **UpdateFileName** when the selection in the list box changes. Attempts to close the dialog box if the selected entry was double-clicked. Clears the selection in the list box when it loses the focus.

See also: **TFileDialog::UpdateFileName**

HandleFName

```
virtual void HandleFName(RTMessage Msg)
  = [ID_FIRST + ID_FNAME];
```

protected

Responds to messages from the edit control. Enables the OK button if the edit control contains text.

SelectFileName

```
void SelectFileName();
```

Selects the text in the edit control. Sets the focus to the edit control.

SetupWindow

```
virtual void SetupWindow();
```

protected

Sets up the file dialog box, calling **TDialog::SetupWindow**. Limits the number of characters that can be entered into the edit control to MAXPATH. Then calls **UpdateListBoxes** and **SelectFileName**.

See also: **TDialog::SetupWindow, TFileDialog::UpdateListBoxes, TFileDialog::SelectFileName**

UpdateFileName

```
void UpdateFileName();
```

Sets the text of the edit control to *PathName* and selects the text.

UpdateListBoxes

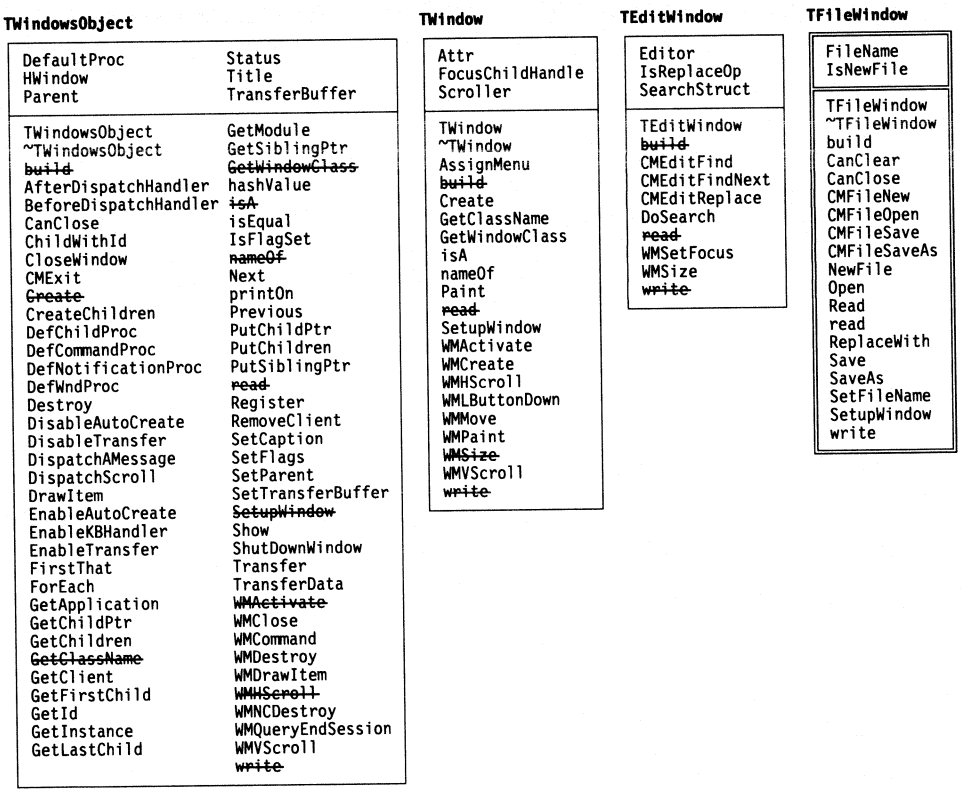
```
BOOL UpdateListBoxes();
```

When opening a file, attempts to update the file and directory list boxes. When saving, attempts to update the directory list box. If updated, calls **UpdateFileName** and returns TRUE. Otherwise, returns FALSE.

See also: **TFileDialog::UpdateFileName**

TFileWindow

filewnd.h



TFileWindow is a file editing window based on **TEditWindow**. File windows use file dialog boxes to open and save files. **TFileWindow**'s data members and member functions manage the file dialog box and automatic responses for typical file commands: Open, Read, Write, Save, and SaveAs.

Data members

FileName

```
LPSTR FileName;
```

Contains the name of the file being edited. It is used by the file dialog box and saving and loading functions.

IsNewFile

```
BOOL IsNewFile;
```

IsNewFile is TRUE if the user is currently editing a new file and FALSE if the user is editing a previously opened file.

Member functions

constructor

```
TFileWindow(PTWindowsObject AParent, LPSTR ATitle, LPSTR AFileName,
            PTModule AModule = NULL);
```

Invokes a **TEditWindow** constructor passing *AParent and ATitle*. Sets *FileName* to a copy of *AFileName* and sets *IsNewFile* to TRUE.

constructor **protected**

```
TFileWindow(StreamableInit);
```

TFileWindow stream constructor. Invokes **TEditWindow**'s stream constructor. Called when a **TFileWindow** is instantiated using data from the supplied stream.

destructor

```
virtual ~TFileWindow();
```

Frees memory allocated to hold the name of the **TFileWindow**.

build

```
static PTStreamable build();
```

Invokes the **TFileWindow(StreamableInit)** constructor. Constructs an object of the type **TFileWindow** prior to reading in its data members from a stream.

CanClear

```
virtual BOOL CanClear();
```

Returns a BOOL value indicating whether it's OK to clear the text of the *Editor*. Returns TRUE if the text has not been changed. If the text has been changed, the user is prompted to save the file or OK clearing the text.

CanClose

```
virtual BOOL CanClose();
```

Returns TRUE if it is OK for the **TFileWindow** to close. Returns the result of a call to **CanClear**.

CMFileNew **protected**

```
virtual void CMFileNew(RTMessage Msg)
  = [CM_FIRST + CM_FILENEW];
```

Responds to an incoming File New command (with a CM_FILENEW command identifier) by calling **NewFile**.

See also: **TFileWindow::NewFile**

CMFileOpen

```
virtual void CMFileOpen(RTMessage Msg)
  = [CM_FIRST + CM_FILEOPEN];
```

protected

Responds to an incoming File Open command (with a CM_FILEOPEN command identifier) by calling **Open**.

See also: **TFileWindow::Open**

CMFileSave

```
virtual void CMFileSave(RTMessage Msg)
  = [CM_FIRST + CM_FILESAVE];
```

protected

Responds to an incoming File Save command (with a CM_FILESAVE command identifier) by calling **Save**.

See also: **TFileWindow::Save**

CMFileSaveAs

```
virtual void CMFileSaveAs(RTMessage Msg)
  = [CM_FIRST + CM_FILESAVEAS];
```

protected

Responds to an incoming File SaveAs command (with a CM_FILESAVEAS command identifier) by calling **SaveAs**.

See also: **TFileWindow::SaveAs**

NewFile

```
void NewFile();
```

Begins the edit of a new file after calling **CanClear** to determine that it is OK to clear the text of the *Editor*.

See also: **TFileWindow::CanClear**

Open

```
void Open();
```

Opens a new file after determining that it is OK to clear the text of the *Editor*. Calls **CanClear**, and if TRUE is returned, brings up a file dialog box to retrieve the name of a new file from the user. Passes the name of the new file by calling **ReplaceWith**.

See also: **TFileWindow::CanClear**, **TFileWindow::ReplaceWith**

Read

```
BOOL Read();
```

Reads the contents of a previously specified file into the *Editor*. Sets *IsNewFile* to FALSE. Returns TRUE if read operation is successful.

read

```
virtual Pvoid read(Ripstream is);
```

protected

Invokes **TEditWindow::read** to read in the base **TEditWindow** object. Then reads in *FileName*. Sets *IsNewFile* to TRUE if *FileName* equals NULL; otherwise, FALSE.

ReplaceWith `void ReplaceWith(LPSTR AFileName);`

Replaces the file currently being edited with a file whose name is supplied, by calling **SetFileName** and **Read**.

See also: **TFileWindow::SetFileName, TFileWindow::Read**

Save `BOOL Save();`

Saves changes to the contents of the *Editor* to a file. But if **Editor->IsModified** returns FALSE, simply returns TRUE, indicating there have been no changes, or the changed contents have already been saved. Otherwise, if *IsNewFile* is TRUE calls **SaveAs**; if *IsNewFile* is FALSE, calls **Write**. Returns the result of either call.

See also: **TFileWindow::SaveAs, TFileWindow::Write**

SaveAs `BOOL SaveAs();`

Saves the contents of the *Editor* to a file whose name is retrieved from the user, through execution of a File Save dialog box. If user selects OK, calls **SetFileName** and **Write**. Returns TRUE if the file was saved.

See also: **TFileWindow::SetFileName, TFileWindow::Write**

SetFileName `void SetFileName(LPSTR AFileName);`

Sets *FileName* and updates the caption of the window.

SetupWindow `virtual void SetupWindow();` **protected**

Creates the edit window's *Editor* edit control by calling **TEditWindow::SetupWindow**. Sets the window's caption to *FileName*, if available, otherwise ("Untitled").

See also: **TFileWindow::SetFileName, TFileWindow::Read**

Write `BOOL Write();`

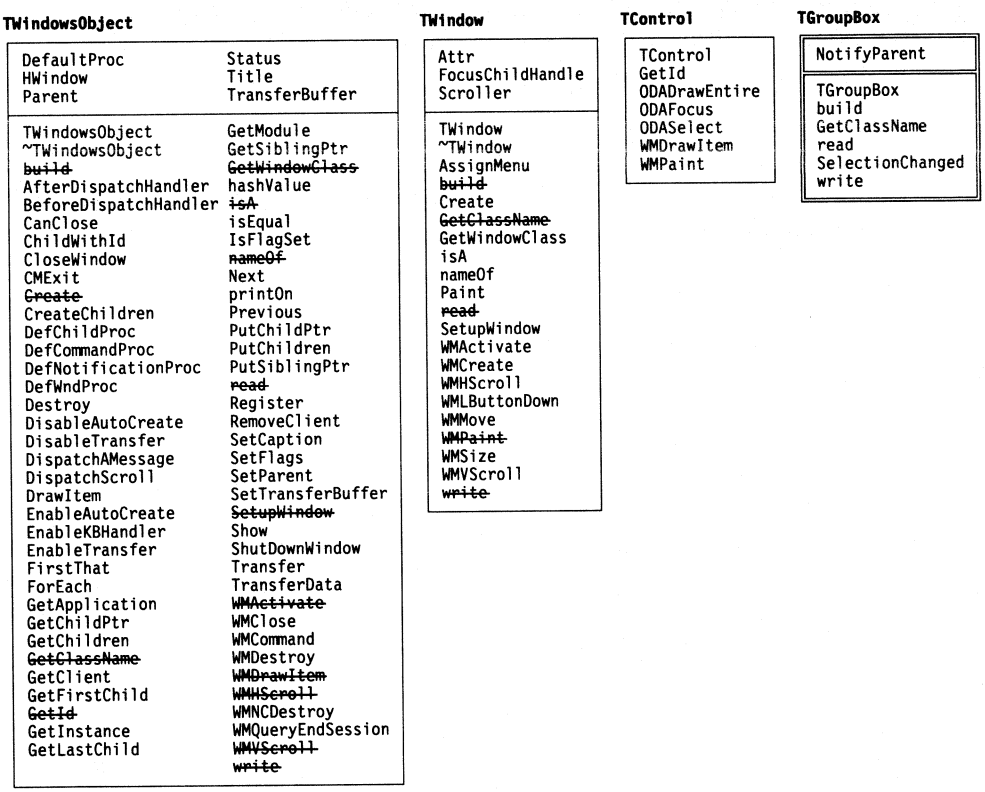
Saves the contents of the *Editor* to a file whose name is specified by *FileName*. Returns TRUE if write operation is successful.

write `virtual void write(Ropstream os);` **protected**

Invokes **TEditWindow::write** to write out the base **TEditWindow** object. Then writes out *FileName*. Returns TRUE if write operation is successful.

TGroupBox

groupbox.h



An instance of a **TGroupBox** is an interface object that represents a corresponding group box element in Windows. Generally, **TGroupBox** objects are not used in dialog boxes or dialog windows (**TDialog**), but are used when you want to create a group box in a window.

While group boxes don't serve an active purpose onscreen, they visually unify a group of selection boxes (check boxes and radio buttons) or other controls. Behind the scenes, however, they can take an important role in handling state changes for their group of controls (normally check boxes or radio buttons).

For example, you may want to respond to a selection change in any one of a group of radio buttons in a similiar manner. You can do this by deriving a class from **TGroupBox** that redefines the member function **SelectionChanged**.

Alternatively, you could respond to selection changes in the group of radio buttons by defining a response for the group box's parent. To do so, define a child-ID-based response member function, using the ID of the group box. The group box will automatically send a child-ID-based message to its parent whenever the radio button selection state changes.

Data member

NotifyParent

```
BOOL NotifyParent;
```

Flag that indicates whether parent is to be notified when the state of the group box's selection boxes has changed. TRUE by default.

Member functions

constructor

```
TGroupBox(PTWindowsObject AParent, int AnId, LPSTR AText, int X, int Y,
          int W, int H, PTModule AModule = NULL);
```

Constructs a group box object with the supplied parent window (*AParent*), control ID (*AnId*), associated text (*AText*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), and height (*H*). Invokes the **TControl** constructor with similar parameters, then modifies *Attr.Style*, adding BS_GROUPBOX and removing WS_TABSTOP. *NotifyParent* is set to TRUE; by default, the group box's parent will be notified when a selection change occurs in any of the group box's controls.

See also: **TControl::TControl**

constructor

```
TGroupBox(PTWindowsObject AParent, int ResourceId,
          PTModule AModule = NULL);
```

Constructs a **TGroupBox** object to be associated with a group box control of a **TDialog**. Invokes the **TControl** constructor with identical parameters. Then calls **DisableTransfer** to exclude group boxes from the transfer mechanism because they have no data to be transferred.

The *ResourceId* parameter must correspond to a group box resource that you define.

See also: **TControl::TControl**, **TWindowsObject::DisableTransfer**

constructor `TGroupBox(StreamableInit);` **protected**

TGroupBox stream constructor. Invokes **TControl**'s stream constructor. Called when a **TGroupBox** is instantiated using data from an input stream.

See also: **TControl::TControl**

build `static PTStreamable build();`

Invokes the **TGroupBox(StreamableInit)** constructor. Constructs an object of the type **TGroupBox** prior to reading in its data members from a stream.

GetClassName `virtual LPSTR GetClassName();` **protected**

Returns the name of **TGroupBox**'s Windows registration class, "BUTTON".

read `virtual Pvoid read(Ripstream is);` **protected**

Invokes **TWindow::read** to read in the base **TGroupBox** object. Then reads in *NotifyParent*.

SelectionChanged `virtual void SelectionChanged(int ControlId);`

If *NotifyParent* is TRUE, notifies the parent window of the group box that one of its selections has changed by sending it a child-ID-based message. This member function could be redefined to allow the group box to handle selection changes in its group of controls.

write `virtual void write(Ropstream os);` **protected**

Invokes **TWindow::write** to write out the base **TGroupBox** object. Then writes out *NotifyParent*.

TInputDialog

inputdia.h

TInputDialog provides a generic dialog box to retrieve text input by a user. When the input dialog box is constructed, its title, prompt, and default input text are specified.

Data members

Buffer `LPSTR Buffer;`

Pointer to the buffer that returns the text retrieved from the user. When passed to the constructor of the input dialog box, contains the default text to be initially displayed in the edit control.

BufferSize `WORD BufferSize;`

Contains the size of the buffer that returns user input.

Prompt `LPSTR Prompt;`

Points to the prompt of the input dialog box.

Member functions

constructor `TInputDialog(PTWindows`

```
Object AParent; LPSTR ATitle, LPSTR APrompt, LPSTR ABuffer,
WORD ABufferSize, PTModule AModule = NULL);
```

Invokes **TDialog**'s constructor, passing it *AParent*, the resource identifier, SD_INPUTDIALOG, and *AModule*. Sets the caption of the dialog box to *ATitle* and the prompt static control to *APrompt*. Sets the *Buffer* and *BufferSize* data members to *ABuffer* and *ABuffersize*.

See also: **TDialog::TDialog**

constructor `TInputDialog(StreamableInit);` **protected**

TInputDialog stream constructor. Invokes **TDialogs**'s stream constructor. Called when a **TInputDialog** is instantiated using data from an input stream.

See also: **TDialog::TDialog**

build `static PTStreamable build();`

Invokes the **TInputDialog(StreamableInit)** constructor. Constructs an object of the type **TInputDialog** prior to reading in its data members from a stream.

read `virtual Pvoid read(Ripstream is);` **protected**

Invokes **TDialog::read** to read in the base **TDialog** object. Then reads in *Prompt*.

SetupWindow `virtual void SetupWindow();` **protected**

In setting up the window, calls **TDialog::SetupWindow**, then limits the number of characters the user can enter to *BufferSize* – 1.

TransferData `void TransferData(WORD Direction);`

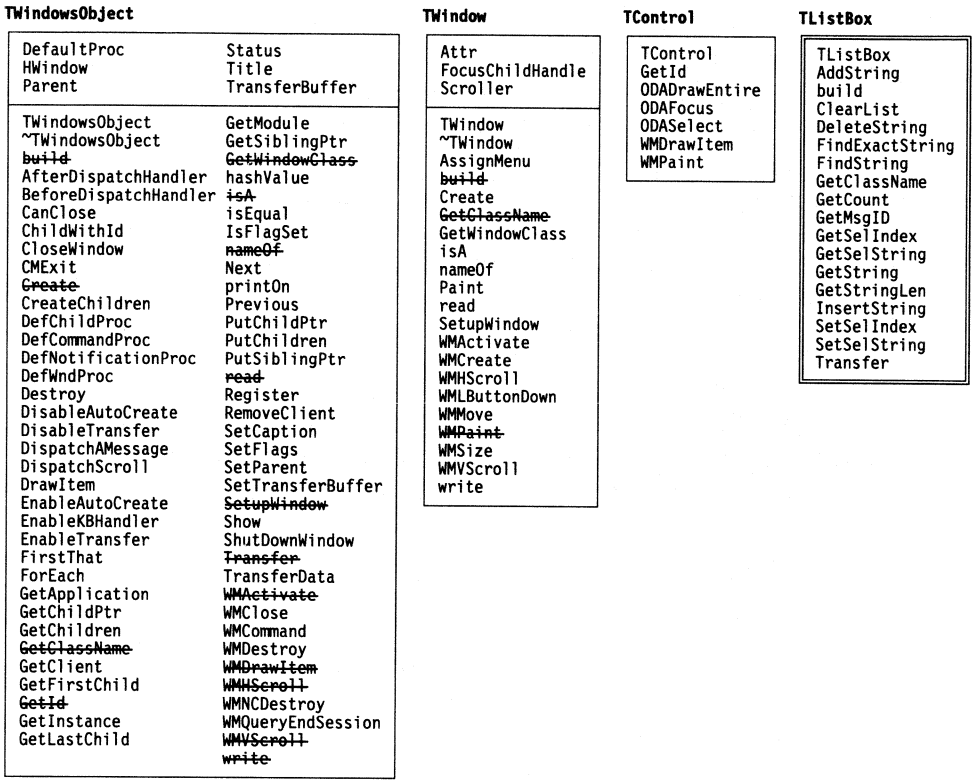
Transfers the data of the input dialog box. If *Direction* is TF_SETDATA, sets the text of the static and edit controls of the dialog box to the text in *Prompt* and *Buffer*. If *Direction* is TF_GETDATA, fills the *Buffer* with the current text of the *Editor*.

write `virtual void write(Ropstream os);` **protected**

Invokes **TDialog::write** to write out the base **TDialog** object. Then writes out *Prompt*.

TListBox

listbox.h



A **TListBox** is an interface object that represents a corresponding list box element in Windows. A **TListBox** must be used to create a list box control in a parent **TWindow**. A **TListBox** can be used to facilitate communication

between your application and the list box controls of a **TDialog**. **TListBox**'s member functions also serve instances of its derived class, **TComboBox**.

Member functions

constructor

```
TListBox(PTWindowsObject AParent, int AnId, int X, int Y, int W, int H,
         PTModule AModule = NULL);
```

Constructs a list box object with the supplied parent window (*AParent*) control ID (*AnId*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), and height (*H*). Invokes a **TControl** constructor. Adds LBS_STANDARD to the default styles for the list box to provide it with

- a border (WS_BORDER)
- a vertical scroll bar (WS_VSCROLL)
- automatic alphabetic sorting of list items (LBS_SORT)
- parent window notification upon selection (LBS_NOTIFY)

The **TListBox** member functions which are described as being for single-selection list boxes are inherited by **TComboBox** and can also be used by combo boxes. Also, these member functions return –1 for multiple-selection list boxes. (See **GetSelIndex**, **GetSelString**, **SetSelIndex**, and **SetSelString**.)

constructor

```
TListBox(PTWindowsObject AParent, int ResourceId,
         PTModule AModule = NULL);
```

Constructs a **TListBox** object to be associated with a list box control of a **TDialog**. Invokes the **TControl** constructor with similar parameters. The *ResourceId* parameter must correspond to a list box resource that you define.

See also: **TControl::TControl**

constructor

```
TListBox(StreamableInit);
```

protected

TListBox stream constructor. Invokes **TControl**'s stream constructor. Called when a **TListBox** is instantiated using data from an input stream.

See also: **TControl::TControl**

AddString

```
int AddString(LPSTR AString);
```

Adds *AString* to the list box, returning its position in the list (0 is the first position). Returns a negative value if an error occurs. The list items are

automatically sorted unless the style LBS_SORT is removed from the list box object's *Attr.Style* data member before creation.

build

```
static PTStreamable build();
```

Invokes the **TListBox(StreamableInit)** constructor. Constructs an object of the type **TListBox** prior to reading in its data members from a stream.

ClearList

```
void ClearList();
```

Clears all items in the list.

DeleteString

```
int DeleteString(int Index);
```

Deletes the item in the list at the position (starting at 0) supplied in *Index*. **DeleteString** returns the number of remaining list items, or a negative value if an error occurs.

FindExactString

```
int FindExactString(LPSTR AString, int SearchIndex);
```

Starting at the line number passed in *SearchIndex*, searches the list box for an exact match with the string *AString*. If a match is not found after the last string has been compared, the search continues from the beginning of the list until a match has been found or until the list has been completely traversed. Searches from the beginning of the list when –1 is supplied as *SearchIndex*. Returns the index of the first string found if successful, a negative value if an error occurs.

FindString

```
int FindString(LPSTR AString, int SearchIndex);
```

Searches the list box as described under **FindExactString**, but looks for the first entry that begins with *AString*.

GetClassName

```
virtual LPSTR GetClassName();
```

protected

Returns the name of **TListBox**'s Windows registration class, "LISTBOX".

GetCount

```
int GetCount();
```

Returns the number of items in the list box, or a negative value if an error occurs.

GetSelString

```
int GetSelString(LPSTR AString, int Index);
```

Retrieves the currently selected item (up to *MaxChars* in length) and returns it in *AString* (includes the terminamting null).

GetSelStrings

```
int GetSelStrings(LPSTR *Strings, int MaxCount, int MaxChars);
```

For multiple-selection list boxes. Retrieves the currently selected items, putting up to *MaxCount* of them in *Strings*. Each entry in the *Strings* array should have space for *MaxChars* characters and a terminating null.

Returns the number of items put into *Strings* (-1 for single-selection list boxes and combo boxes).

GetMsgID `virtual WORD GetMsgID(WORD AMsg);` **protected**

Returns the identifier of the Windows list box message associated with the specified ObjectWindows message identifier. Allows **TComboBox**es (and derived classes) to inherit many **TListBox** member functions.

GetSelCount `int GetSelCount();`

Returns the number of selected items in the list box. For single- or multiple-selection list boxes (and combo boxes).

GetSelIndex `int GetSelIndex();`

Returns the non-negative index (starting at 0) of the currently selected item, or a negative value if no item is selected. For single-selection list boxes.

GetSelIndexes `int GetSelIndexes(Pint Indexes, int MaxCount);`

For multiple-selection list boxes. Fills the *Indexes* array with the indexes of up to *MaxCount* selected strings. Returns the number of items put in *Indexes* (-1 for single-selection list boxes and combo boxes).

GetSelString `int GetSelString(LPSTR AString, int MaxChars);`

Retrieves the currently selected item, as long as it is no longer than *MaxChars* and returns it in *AString* (includes a terminating NULL). **GetSelString** returns the string length, a negative value if an error occurs, or –1 if no string is selected. For single-selection list boxes.

GetString `int GetString(LPSTR AString, int Index);`

Retrieves the item at the position (starting at 0) supplied in *Index* and returns it in *AString*. **GetString** returns the string length, or a negative value if an error occurs.

GetStringLen `int GetStringLen(int Index);`

Returns the string length (excluding the terminating NULL) of the item at the position index supplied in *Index*. Returns a negative value in the case of an error.

InsertString `int InsertString(LPSTR AString, int Index);`

Inserts *AString* in the list box at the position supplied in *Index*, and returns the item's actual position (starting at 0) in the list. A negative value is returned if an error occurs. The list is not resorted. If *Index* is –1, the string is appended to the end of the list.

SetSelIndex

```
int SetSelIndex(int Index);
```

Forces the selection of the item at the position (starting at 0) supplied in *Index*. If *Index* is –1, the list box is cleared of any selection. **SetSelIndex** returns a negative number if an error occurs. For single-selection list boxes.

GetSelIndexes

```
int SetSelIndexes(Pint Indexes, int NumSelections, BOOL ShouldSet);
```

For multiple-selection list boxes. Selects/deselects the strings in the associated list box at the indexes specified in the *Indexes* array. If *ShouldSet* is TRUE, the indexed strings are selected and highlighted; if *ShouldSet* is FALSE the highlight is removed and they are no longer selected. Returns the number of strings successfully selected or deselected (-1 for single-selection list boxes and combo boxes). If *NumSelections* is less than zero, all strings are selected or deselected, and a negative value is returned on failure.

SetSelString

```
int SetSelString(LPSTR AString, int AIndex);
```

Forces the selection of the first item beginning with the text supplied in *AString* that appears beyond the position (starting at 0) supplied in *AIndex*. If *AIndex* is –1, the entire list is searched, beginning with the first item. **SetSelString** returns the position of the newly selected item, or a negative value in the case of an error. For single-selection list boxes.

SetSelStrings

```
int SetSelStrings(LPSTR *Prefixes, int NumSelections, BOOL ShouldSet);
```

For multiple-selection list boxes. Selects the strings in the associated list box which begin with the prefixes specified in the *Prefixes* array. For each string the search begins at the beginning of the list and continues until a match is found or until the list has been completely traversed. If *ShouldSet* is TRUE, the matched strings are selected and highlighted; if *ShouldSet* is FALSE the highlight is removed from the matched strings and they are no longer selected. Returns the number of strings successfully selected or deselected (-1 for single-selection list boxes and combo boxes). If *NumSelections* is less than zero, all strings are selected or deselected, and a negative value is returned on failure.

Transfer

```
virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);
```

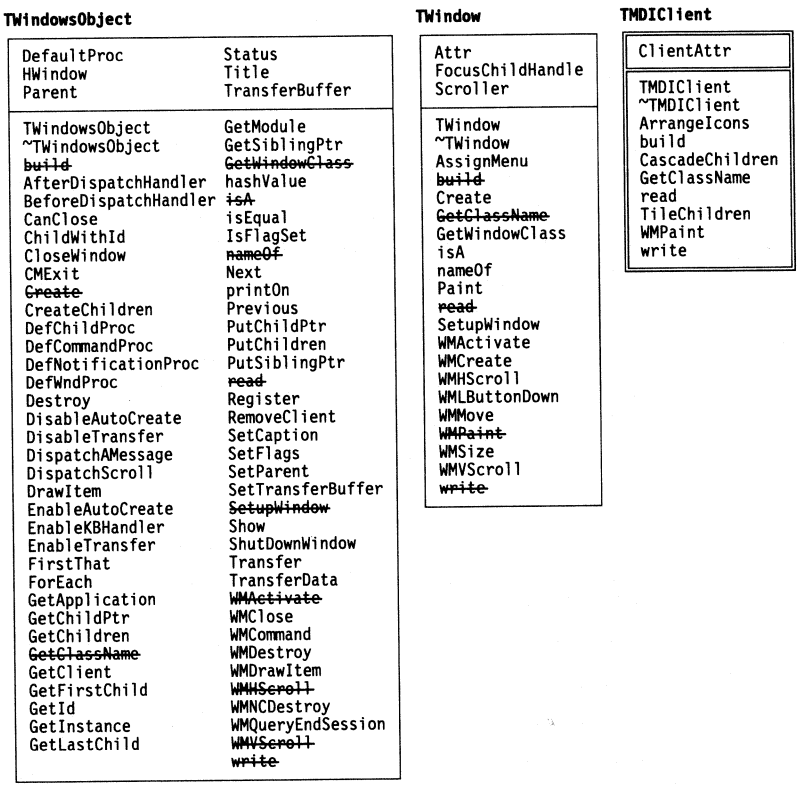
Transfers the items and selection(s) of the list box to or from a transfer buffer if TF_SETDATA or TF_GETDATA, respectively, is passed as the *TransferFlag*. *TransferBuffer* is expected to point to a **PTListBoxData** object which is used as a data transfer buffer for a list box.

Transfer returns the size of **PTListBoxData** (the pointer, not the structure). To retrieve the size without transferring data, pass TF_SIZEDATA as the *TransferFlag*.

You must use a pointer in your transfer buffer to these structures. You cannot imbed copies of the structures in your transfer buffer. And you can't use these structures as transfer buffers.

See also: **TListBoxData, TWindowsObject::TransferBuffer**

TMDIClient — mdi.h



Multiple Document Interface (MDI) client windows (represented by a **TMDIClient** object) manage the MDI child windows of a **TMDIFrame** parent.

Data member

ClientAttr

```
LPCLIENTCREATESTRUCT ClientAttr;
```

ClientAttr holds a pointer to a structure of the MDI client window's attributes.

Member functions

constructor

```
TMDIClient(PTMDIFrame AParent, PTModule AModule = NULL);
```

Constructs a MDI client window object. Invokes a **TWindow** constructor passing *AParent* and *AModule*. Sets *Attr.Id* to the default client window identifier, ID_MDICLIENT. Sets *Attr.Style* to WS_CHILD | WS_VISIBLE | WS_GROUP | WS_TABSTOP | WS_CLIPCHILDREN. Initializes *ClientAttr*, setting its idFirstChild member to ID_FIRSTMDICHILD.

See also: **TWindow::TWindow**

constructor

```
TMDIClient(PTMDIFrame AParent, HWND AnHWindow, PTModule AModule = NULL);
```

Constructs an MDI client object to be associated with the MDI client window whose handle is specified. When invoking **TWindow**'s constructor, passes *AHWindow* and *AModule*. Can be used to construct MDI client windows in DLLs where the parent window is a non-ObjectWindows window.

constructor

```
TMDIClient(StreamableInit);
```

protected

TMDIClient stream constructor. Invokes **TWindow**'s stream constructor. Called when a **TMDIClient** is instantiated using data from an input stream.

See also: **TWindow::TWindow**

destructor

```
virtual ~TMDIClient();
```

Frees the *ClientAttr* structure, if present.

See also: **TWindow::~TWindow**

ArrangeIcons

```
virtual void ArrangeIcons();
```

Neatly arranges the MDI child window icons at the bottom of the MDI client window.

build `static PTStreamable build();`

Invokes the **TMDIClient(StreamableInit)** constructor. Constructs an object of the type **TMDIClient** prior to reading in its data members from a stream.

CascadeChildren `virtual void CascadeChildren();`

Sizes and arranges all of the non-iconized MDI child windows within the MDI client window. The children are overlapped, although each title bar is visible.

GetClassName `virtual LPSTR GetClassName();` **protected**

Returns **TMDIClient**'s Windows registration class name, "MDICLIENT".

read `virtual Pvoid read(Ripstream is);` **protected**

Invokes **TWindow::read** to read in the base **TWindow** object. Sets *ClientAttr->hWindowMenu* to 0. Reads in *ClientAttr->idFirstChild*.

TileChildren `virtual void TileChildren();`

Sizes and arranges all of the non-iconized MDI child windows within the MDI client window. The children fill up the entire client area without overlapping.

WMMDIActivate `virtual void WMMDIActivate (RTMessage) = [WM_FIRST + WM_MDIACTIVATE];` **protected**

Responds to an incoming WM_MDIACTIVATE message. Overrides **TWindow::WMMDIActivate** and instead calls **DefWndProc**. If you redefine **WMMDIActivate** in a derived class, be sure to invoke the **WMMDIActivate** of the base class after any other processing (unless you have a special reason for invoking it previously).

WMPaint `virtual void WMPaint(RTMessage Msg) = [WM_FIRST + WM_PAINT];` **protected**

Redefines **TWindow::WMPaint** to call **DefWndProc**.

See also: **TWindow::Paint, TScroller::BeginView, TScroller::EndView**

write `virtual void write(Ropstream os);` **protected**

Invokes **TWindow::write** to write out the base **TWindow** object. Then writes out *ClientAttr->idFirstChild*.

TMDIFrame

mdi.h

Multiple Document Interface (MDI) frame windows, represented by **TMDIFrame**, are overlapped windows that serve as main windows of MDI-compliant applications. **TMDIFrame** objects automatically handle the creation and initialization of an MDI client "window" (represented by a **TMDIClient** object) required by Windows. **TMDIFrame**s also supply member functions that manipulate MDI child windows, such as **TileChildren** and **CloseChildren**.

Data members

ActiveChild

```
PTWindow ActiveChild;
```

ActiveChild points to the **TMDIFrame**'s active MDI child window. *ActiveChild* is set by the child in its **WMMDIActivate** message response

member function. **TMDIFrame**'s constructors initialize *ActiveChild* and it is read and written by the **read** and **write** functions.

ChildMenuPos

```
int ChildMenuPos;
```

ChildMenuPos contains the position in the MDI window's top-level menu of the child window submenu. The zero position is at top left.

ClientWnd

```
PTMDIClient ClientWnd;
```

ClientWnd points to the **TMDIFrame**'s client window, an instance of **TMDIClient**.

Member functions

constructor

```
TMDIFrame(LPSTR ATitle, int MenuId, PTModule AModule = NULL);
```

Constructs an MDI frame window object using the caption (*ATitle*) and the integer resource identifier (*MenuId*).

constructor

```
TMDIFrame(LPSTR ATitle, LPSTR MenuName, PTModule AModule = NULL);
```

Constructs an MDI frame window object using the caption (*ATitle*) and the string resource identifier (*MenuName*). (MDI frame windows are required to have menus, and must not have a parent.) As a default, sets *ChildMenuPos* to 0, indicating that the child window menu is at the top left position. You can reset *ChildMenuPos* in the constructor of your derived type. For example,

```
TMyMDIFrame::TMyMDIFrame(LPSTR ATitle, LPSTR MenuName,
  PTModule AModule = NULL) : TMDIFrame(ATitle, AMenu, AModule)
{
   ChildMenuPos = 3;
}
```

constructor

```
TMDIFrame(HWND AnHWindow, HWND ClientWnd);
```

Constructs an MDI window object to be associated with the MDI window whose handle is specified. Invokes a **TWindow** constructor, then constructs an MDI client object to be associated with the client window whose handle is specified.

constructor

```
TMDIFrame(StreamableInit);
```

protected

TMDIFrame stream constructor. Invokes **TWindow**'s stream constructor. Called when a **TMDIFrame** is instantiated using data from an input stream.

See also: **TWindow::TWindow**

destructor

```
virtual ~TMDIFrame();
```

Deletes the MDI client window object.

See also: **TWindow::~TWindow**

ArrangeIcons

```
virtual void ArrangeIcons();
```

Arranges the iconized MDI child windows into a neat row at the bottom of the MDI client window by calling the **ArrangeIcons** member function of its client window.

See also: **TMDIClient::ArrangeIcons**

build

```
static PTStreamable build();
```

Invokes the **TMDIFrame(StreamableInit)** constructor. Constructs an object of the type **TMDIFrame** prior to reading in its data members from a stream.

CascadeChildren

```
virtual void CascadeChildren();
```

Sizes and arranges all of the non-iconized MDI child windows within the MDI client window by calling the **CascadeChildren** member function of its client window.

See also: **TMDIClient::CascadeChildren**

CloseChildren

```
virtual BOOL CloseChildren();
```

Shuts down all MDI child windows if all can be closed. Returns TRUE if all children are closed (or there were no children); returns FALSE if any child can't be closed.

See also: **TWindow::CanClose**

CMArrangeIcons

```
virtual void CMArrangeIcons(RTMessage Msg)
  = [CM_FIRST + CM_ARRANGEICONS];
```

protected

Responds to a menu selection with an ID of CM_ARRANGEICONS by calling **ArrangeIcons**.

See also: **TMDIFrame::ArrangeIcons**

:MCascadeChildren

```
virtual void CMCascadeChildren(RTMessage Msg)
  = [CM_FIRST + CM_CASCADECHILDREN];
```

protected

Responds to a menu selection with an ID of CM_CASCADECHILDREN by calling **CascadeChildren**.

See also: **TMDIFrame::CascadeChildren**

CMCloseChildren

```
virtual void CMCloseChildren(RTMessage Msg)
  = [CM_FIRST + CM_CLOSECHILDREN];
```

protected

Responds to a menu selection with an ID of CM_CLOSECHILDREN by calling **CloseChildren**.

See also: **TMDIFrame::CloseChildren**

CMCreateChild

```
virtual void CMCreateChild(RTMessage Msg)
  = [CM_FIRST + CM_CREATECHILD];
```

protected

Responds to a menu selection with a menu ID of CM_CREATECHILD by calling **CreateChild** to produce a new child window.

See also: **TMDIFrame::CreateChild**

CMTileChildren

```
virtual void CMTileChildren(RTMessage Msg)
  = [CM_FIRST + CM_TILECHILDREN];
```

protected

Responds to a menu selection with an ID of CM_TILECHILDREN by calling **TileChildren**.

See also: **TMDIFrame::TileChildren**

CreateChild

```
virtual PTWindowsObject CreateChild();
```

Constructs and creates a new MDI child window by calling **InitChild** and **MakeWindow**. Returns a pointer to the new MDI child window.

See also: **TMDIFrame::InitChild, TModule::MakeWindow**

GetClassName

```
virtual LPSTR GetClassName();
```

protected

Returns the name of **TMDIFrame**'s Windows registration class, "OWLMDIFrame".

GetClient

```
virtual PTMDIClient GetClient();
```

Returns a pointer to the MDI client window stored in **ClientWnd**.

GetWindowClass

```
virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);
```

protected

Modifies the default window class structure and passes it back in *AWndClass*. Calls **TWindow::GetWindowClass**, then sets *AWndClass.Style* to 0.

See also: **TWindow::GetWindowClass**

InitChild

```
virtual PTWindowsObject InitChild();
```

Constructs an instance of **TWindow** as an MDI child window and returns a pointer to it. Redefine this member function in your derived MDI

window class to construct an instance of a derived MDI child class. For example,

```
PTWindowsObject TMyMDIFrame::InitChild()
{
   return new TMyMDIChild(this, "");
}
```

See also: **TMDIFrame::CreateChild**

InitClientWindow

```
virtual void InitClientWindow();
```

Constructs the MDI client window as an instance of **TMDIClient**, storing its pointer in *ClientWnd*.

read

```
virtual Pvoid read(Ripstream is);
```

protected

Invokes **TWindow::read** to read in the base **TWindow** object. Then reads in *ClientWnd* and *ChildMenuPos*.

SetupWindow

```
virtual void SetupWindow();
```

protected

Calls **InitClientWindow** to construct a MDI client window. Removes the MDI client window from the child list; it is processed separately. If you redefine **SetupWindow** in a derived type, be sure to explicitly call **TMDIFrame::SetupWindow**.

See also: **TMDIFrame::InitClientWindow**

TileChildren

```
virtual void TileChildren();
```

Sizes and arranges all of the non-iconized MDI child windows within the MDI client window by calling the **TileChildren** member function of its client window.

See also: **TMDIClient::TileChildren**

WMActivate

```
virtual void WMActivate(RTMessage Msg) = [WM_FIRST +
WM_ACTIVATE];
```

protected

Responds to an incoming WM_ACTIVATE message. Since an MDI child doesn't get WM_MDIACTIVATE messages when its frame window gets activated and deactivated, the active child's **ActivationResponse** member function is called here.

See also: **TWindow::ActivationResponse, TWindowsObject::WMActivate**

write

```
virtual void write(Ropstream os);
```

protected

Invokes **TWindow::write** to write out the base **TWindow** object. Then writes out *ClientWnd* and *ChildMenuPos*.

TModule — module.h

ObjectWindows dynamic-link libraries (DLLs) construct an instance of **TModule**, which acts as an object-oriented stand-in for the library (DLL) module. ObjectWindows applications construct an instance of **TApplication**, derived from **TModule**. **TModule** defines behavior shared by both library and application modules. **TModule** member functions provide support for window memory management and error-processing.

Data members

hInstance

```
HANDLE hInstance;
```

Contains the handle of the executing instance of either the Windows application or DLL module. The handle must be supplied as a parameter to Windows when loading resources.

lpCmdLine

```
LPSTR lpCmdLine;
```

Points to a copy of the command line passed when the module was loaded. Is a null-terminated string.

Name

```
LPSTR Name;
```

Name holds the name of the application or DLL module. This can be used internally by the module's code.

Status

```
int Status;
```

Status contains the module status. It can be set by ObjectWindows to indicate an error in creating the main window of an application module (as the **WinMain** return value). Or it can be set to indicate an error in the startup of a (DLL) library module (uses *Status* to determine the value which **LibMain** returns).

Member functions

constructor

```
TModule(LPSTR AName, HANDLE AnInstance, LPSTR ACmdLine);
```

Initializes the data members using the specified parameters. Makes a copy of *ACmdLine* and sets *lpCmdLine* to the allocated string. Sets *Status* to EM_INVALIDMODULE if an error occurs.

destructor

```
virtual ~TModule();
```

Destructor for a **TModule** object, deletes *lpCmdLine*.

Error `virtual void Error(int ErrorCode);`

Error processes errors identified by the error value supplied in *ErrorCode*. *ErrorCode* can be an error you define, or one of the EM_ constants described in Chapter 18, "Miscellaneous components."

Error displays the error code in a message box and asks the user if it is OK to proceed. If not, program execution stops (the program is terminated). If you continue, the program may or may not be able to recover.

ExecDialog `virtual int ExecDialog(PTWindowsObject ADialog);`

Executes a modal dialog box, if the supplied dialog box object is valid. Calls **ValidWindow** to determine the validity of the object. If valid, calls the dialog box's **Execute** member function and returns the result of the call. If not valid, returns IDCANCEL. Invalid objects and objects whose execution fails (*Status* is non-zero) are deleted by **ExecDialog**, otherwise they are deleted by **Execute**.

See also: **TModule::ValidWindow, TDialog::Execute, TModule::MakeWindow**

GetClientHandle `HWND GetClientHandle(HWND AnHWindow);`

Returns the handle of the MDI client window of the window whose handle is specified. Returns 0 if the window does not have an MDI client window.

GetParentObject `virtual PTWindowsObject GetParentObject(HWND ParentHandle);`

Called in a DLL to obtain a pointer to a parent window object. If the caller is an ObjectWindows application, **GetParentObject** returns a pointer to the parent window object in the caller. For a non-ObjectWindows caller, **GetParentObject** returns a pointer to an object that it constructs to function as an object in the DLL for the parent window in the calling application.

hashValue `virtual hashValueType hashValue() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *hInstance*, the hash value of a **TModule**.

isA `virtual classType isA() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *moduleClass*, the class identifier of **TModule**.

isEqual `virtual int isEqual(RCObject module) const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns TRUE if **this** and *module* have equal *hInstance* values.

LowMemory `BOOL LowMemory();`

Returns TRUE if the safety pool is exhausted.

See also: **TModule::RestoreMemory**

MakeWindow `virtual PTWindowsObject MakeWindow(PTWindowsObject AWindowsObject);`

Creates a window or modeless dialog box element to be associated with the supplied object, if the object is valid. Calls **ValidWindow** to determine the validity of the object. If valid, calls the object's **Create** member function. If **Create** returns TRUE, returns *AWindowsObject*.

Invalid objects and objects that cannot be created are deleted before **MakeWindow** returns, returning NULL.

See also: **TModule::ValidWindow, TWindow::Create**

nameOf `virtual Pchar nameOf() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns "TModule", the class identification string of **TModule**.

printOn `virtual void printOn(Rostream outputStream) const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Prints a hexadecimal representation of the module's instance handle (*hInstance*) on the specified stream.

RestoreMemory `void RestoreMemory();`

Allocates a safety pool (if not already allocated).

See also: **SafetyPool::Allocate, TModule::LowMemory**

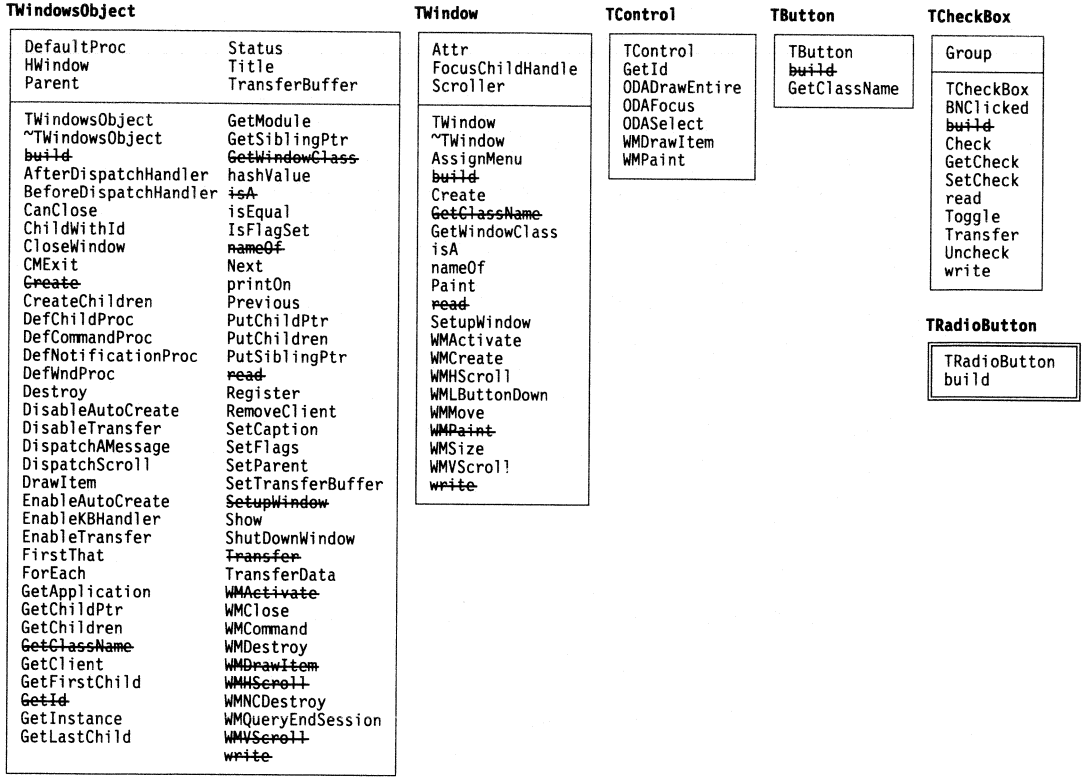
ValidWindow `virtual PTWindowsObject ValidWindow(PTWindowsObject AWindowsObject);`

Determines whether *AWindowsObject* is a valid object. If *AWindowsObject* is valid, **ValidWindow** returns a pointer to it; otherwise, it deletes the object and returns NULL. *AWindowsObject* is invalid if either of two conditions occurs: **LowMemory** returns TRUE (allocation of the object endangered the safety pool), or the *Status* data member of *AWindowsObject* is nonzero.

See also: **TModule::LowMemory**, *TWindowsObject::Status*

TRadioButton

radiobut.h



A **TRadioButton** is an interface object that represents a corresponding radio button element in Windows. Use **TRadioButton** to create a radio button control in a parent **TWindow**. A **TRadioButton** can also be used to facilitate communication between your application and the radio button controls of a **TDialog**.

Radio buttons have two states: checked and unchecked. **TRadioButton** inherits its state management member functions from its base class, **TCheckBox**. Optionally, a radio button can be part of a group (**TGroupBox**) that visually and logically groups its controls.

Member functions

constructor

```
TRadioButton(PTWindowsObject AParent, int AnId, LPSTR ATitle,
             int X, int Y, int W, int H, PTGroupBox AGroup,
             PTModule AModule = NULL);
```

Constructs a radio button object with the supplied parent window (*AParent*), control ID (*AnId*), associated text (*ATitle*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), and associated group box (*AGroup*). Invokes the **TCheckBox** constructor with similar parameters. Then sets the *Attr.Style* data member to WS_CHILD | WS_VISIBLE | BS_AUTORADIOBUTTON.

See also: **TControl::TControl**

constructor

```
TRadioButton(PTWindowsObject AParent, int ResourceId, PTGroupBox AGroup,
             PTModule AModule = NULL);
```

Constructs a **TRadioButton** object to be associated with a radio button control of a **TDialog**. Invokes the **TCheckBox** constructor with identical parameters. The *ResourceId* parameter must correspond to a radio button resource that you define.

constructor **protected**

```
TRadioButton(StreamableInit);
```

TRadioButton stream constructor. Invokes **TCheckBox**'s stream constructor. Called when a **TRadioButton** is instantiated using data from an input stream.

See also: **TCheckBox::TCheckBox**

BNClicked **protected**

```
virtual void BNClicked(RTMessage Msg) = [NF_FIRST +
BN_CLICKED];
```

Responds to an incoming BN_CLICKED message.

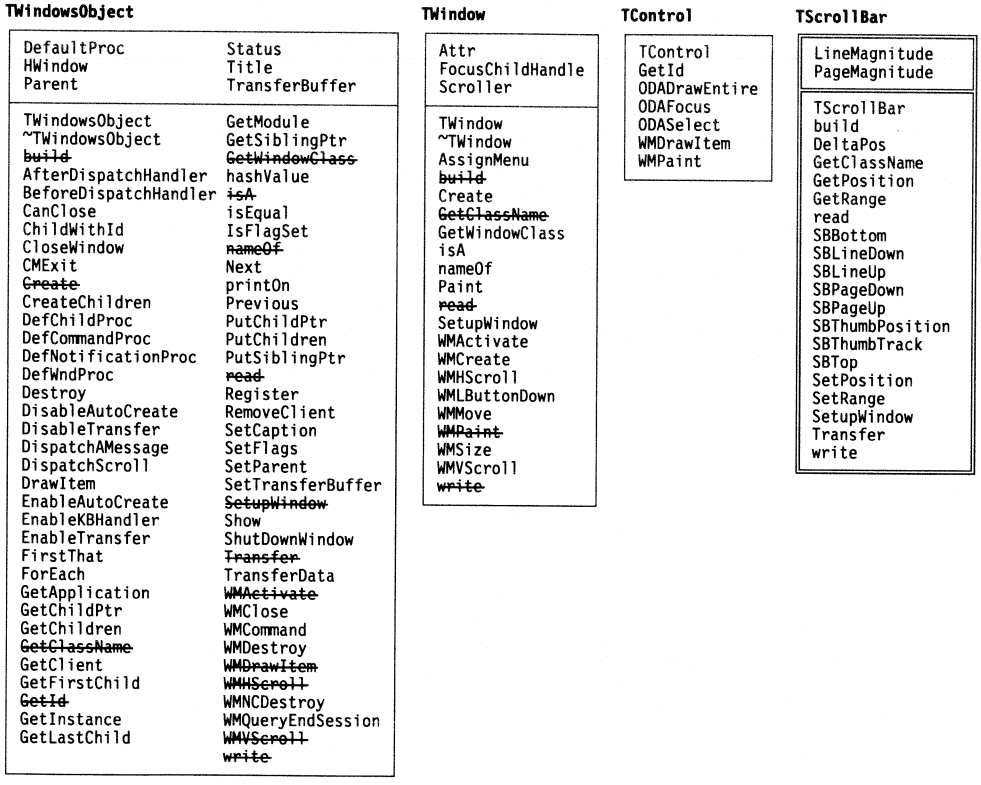
build

```
static PTStreamable build();
```

Invokes the **TRadioButton(StreamableInit)** constructor. Constructs an object of the type **TRadioButton** prior to reading in its data members from a stream.

TScrollBar

scrollba.h



TScrollBar objects represent standalone vertical and horizontal scroll bar controls. Most of **TScrollBar**'s member functions are concerned with managing the scroll bar's sliding box (thumb) position and range.

One special feature of the type **TScrollBar** is the notify-based set of member functions that automatically adjust the scroll bar's thumb position in response to Windows scroll bar messages.

Never place **TScrollBar** objects in windows that have either the WS_HSCROLL or WS_VSCROLL styles in their attributes.

Data members

LineMagnitude

```
int LineMagnitude;
```

LineMagnitude is the number of range units to scroll the scroll bar when the user requests a small movement by clicking on the scroll bar's arrows. **TScrollBar**'s constructor sets *LineMagnitude* to 1 by default. (The scroll range is 0-100 by default.)

See also: **TScrollBar::SetupWindow**

PageMagnitude

```
int PageMagnitude;
```

PageMagnitude is the number of range units to scroll the scroll bar when the user requests a large movement by clicking in the scroll bar's scrolling area. **TScrollBar**'s constructor sets *PageMagnitude* to 10 by default. (The scroll range is 0-100 by default.)

Member functions

constructor

```
TScrollBar(PTWindowsObject AParent, int AnId, int X, int Y, int W, int H,
           BOOL IsHScrollBar, PTModule AModule = NULL);
```

Constructs and initializes a **TScrollBar** object with the given parent window (*AParent*), a control ID (*AnId*), a position (*X*, *Y*), and a size of (*W*, *H*). Invokes the **TControl** constructor with similar parameters. If *IsHScrollBar* is TRUE, adds SBS_HORZ to the styles specified in *Attr.Style*. If not TRUE, adds SBS_VERT. If the supplied height for a horizontal scroll bar or the supplied width for a vertical scroll bar is 0, a standard value is used. *LineMagnitude* is initialized to 1 and *PageMagnitude* is set to 10.

See also: **TControl::TControl**

constructor

```
TScrollBar(PTWindowsObject AParent, int ResourceId,
           PTModule AModule = NULL);
```

Constructs a **TScrollBar** object to be associated with a scroll bar control of a **TDialog**. Invokes the **TControl** constructor with identical parameters.

The *ResourceId* parameter must correspond to a scroll bar resource that you define.

constructor `TScrollBar(StreamableInit);` **protected**

TScrollBar stream constructor. Invokes **TControl**'s stream constructor. Called when a **TScrollBar** is instantiated using data from an input stream.

See also: **TControl::TControl**

build `static PTStreamable build();`

Invokes the **TScrollBar(StreamableInit)** constructor. Constructs an object of the type **TScrollBar** prior to reading in its data members from a stream.

DeltaPos `int DeltaPos(int Delta);`

Calls **SetPosition** to change the scroll bar's thumb position by the value supplied in *Delta*. A positive *Delta* moves the thumb down or right. A negative *Delta* value moves the thumb up or left. The new thumb position is returned.

See also: **TScrollBar::SetPosition**

GetClassName `virtual LPSTR GetClassName();` **protected**

Returns the name of **TScrollBar**'s Windows registration class, "SCROLLBAR".

GetPosition `int GetPosition();`

Returns the scroll bar's current thumb position.

See also: **TScrollBar::SetPosition**

GetRange `void GetRange(Rint LoVal, Rint HiVal);`

Returns the end values of the present range of scroll bar thumb positions in *LoVal* and *HiVal*.

See also: **TScrollBar::SetPosition**, **TScrollBar::SetRange**

read `virtual Pvoid read(Ripstream is);` **protected**

Invokes **TWindow::read** to read in the base **TWindow** object. Then reads in *LineMagnitude*, and *PageMagnitude*.

SBBottom
```
virtual void SBBottom(RTMessage Msg)
  = [NF_FIRST + SB_BOTTOM];
```
protected

Moves the thumb to the bottom or right of the scroll bar by calling **SetPosition**. This member function is called to respond to the thumb being dragged to the bottom or rightmost position of the scroll bar.

See also: **TScrollBar::SetPosition**

SBLineDown

```
virtual void SBLineDown(RTMessage Msg)
  = [NF_FIRST + SB_LINEDOWN];
```

protected

Moves the thumb down or right (by *LineMagnitude* units) by calling **SetPosition**. This member function is called to respond to a click on the bottom or right arrow of the scroll bar.

See also: **TScrollBar::SetPosition**

SBLineUp

```
virtual void SBLineUp(RTMessage Msg)
  = [NF_FIRST + SB_LINEUP];
```

protected

Moves the thumb up or left (by *LineMagnitude* units) by calling **SetPosition**. This member function is called to respond to a click on the top or left arrow of the scroll bar.

See also: **TScrollBar::SetPosition**

SBPageDown

```
virtual void SBPageDown(RTMessage Msg)
  = [NF_FIRST + SB_PAGEDOWN];
```

protected

Moves the thumb down or right (by *PageMagnitude* units) by calling **SetPosition**. This member function is called to respond to a click in the bottom or right scrolling area of the scroll bar.

See also: **TScrollBar::SetPosition**

SBPageUp

```
virtual void SBPageUp(RTMessage Msg)
  = [NF_FIRST + SB_PAGEUP];
```

protected

Moves the thumb up or left (by *PageMagnitude* units) by calling **SetPosition**. This member function is called to respond to a click in the top or left scrolling area of the scroll bar.

See also: **TScrollBar::SetPosition**

SBThumbPosition

```
virtual void SBThumbPosition(RTMessage Msg)
  = [NF_FIRST + SB_THUMBPOSITION];
```

protected

Moves the thumb by calling **SetPosition**. This member function is called to respond to the thumb being set to a new position.

See also: **TScrollBar::SetPosition**

SBThumbTrack

```
virtual void SBThumbTrack(RTMessage Msg)
  = [NF_FIRST + SB_THUMBTRACK];
```

protected

Moves the thumb as it is being dragged to a new position, by calling **SetPosition**.

See also: **TScrollBar::SetPosition**

SBTop `virtual void SBTop(RTMessage Msg) = [NF_FIRST + SB_TOP];` **protected**

Moves the thumb to the top or right of the scroll bar by calling **SetPosition**. This member function is called to respond to the thumb being dragged to the top or right-most position on the scroll bar.

See also: **TScrollBar::SetPosition**

SetPosition `void SetPosition(int ThumbPos);`

Moves the thumb to the position specified in *ThumbPos*. If *ThumbPos* is outside the present range of the scroll bar, the thumb is moved to the closest position within range.

See also: **TScrollBar::GetPosition**

SetRange `void SetRange(int LoVal, int HiVal);`

Sets the scroll bar to the range between *LoVal* and *HiVal*.

See also: **TScrollBar::GetRange**

SetupWindow `virtual void SetupWindow();` **protected**

Sets the scroll bar's range to 0, 100. To redefine this range, call **SetRange**.

See also: **TScrollBar::SetRange**

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

Transfers scroll bar data (which contains the low and high values of the range and the thumb position) to or from the transfer buffer pointed to by *DataPtr*. *DataPtr* is expected to point to a **TScrollBarData** structure, defined in Chapter 18, "Miscellaneous components."

Data is transferred to or from the transfer buffer if TF_GETDATA or TF_SETDATA is supplied as the *TransferFlag*.

Transfer always returns the size of the transfer data (the size of the **TScrollBarData** structure). To retrieve the size of this data without transferring data, pass TF_SIZEDATA as the *TransferFlag*.

write `virtual void write(Ropstream os);` **protected**

Invokes **TWindow::write** to write out the base **TWindow** object. Then writes out *LineMagnitude*, and *PageMagnitude*.

TScroller — scroller.h

TScroller instances aid scrolling of a **TWindow**'s display, usually in conjunction with the window's scroll bars. In contrast, auto-scrolling scrolls a window's display in response to a mouse drag from within the window to without.

To utilize **TScroller** functionality, set the *Scroller* member of your **TWindow** descendant to a **TScroller** object instantiated in the constructor of your **TWindow** descendant.

Data members

AutoMode

```
BOOL AutoMode;
```

AutoMode is TRUE if "auto-scrolling" is in effect for the **TScroller**. When *AutoMode* is TRUE, the owner window's display is scrolled in response to a mouse drag from within the window to without. By default, *AutoMode* is TRUE.

AutoOrg

```
BOOL AutoOrg;
```

AutoOrg is a flag (TRUE by default) indicating that the scroller should automatically offset the origin of the client area by *XPos*, *YPos* when setting up for painting.

See also: **TScroller::BeginView**

ClassHashValue

```
static hashValueType ClassHashValue;
```

protected

Contains the hash value of the last constructed instance of **TScroller**. It is used as a unique key for **TScroller** instances.

HasHScrollBar

```
BOOL HasHScrollBar;
```

If the owner window has a horizontal window scroll bar, *HasHScrollBar* is TRUE.

HasVScrollBar

```
BOOL HasVScrollBar;
```

If the owner window has a vertical window scroll bar, *HasVScrollBar* is TRUE.

InstanceHashValue

```
hashValueType InstanceHashValue;
```

protected

Contains the hash value of **this**. Provides a unique key for **hashValue**.

TrackMode

```
BOOL TrackMode;
```

TrackMode is TRUE if the owner window's display should be scrolled as the thumb is being dragged. If FALSE, the display will not be scrolled until the thumb drag is completed. By default, *TrackMode* is TRUE.

Window `PTWindow Window;`

Window points to the **TScroller**'s owner window.

XLine `int XLine;`

XLine is the number of *XUnit*s to scroll horizontally in response to a click on the horizontal scroll bar arrow. The default value is 1.

XPage `int XPage;`

XPage is the number of *XUnit*s to scroll horizontally in response to a click on the horizontal scroll bar's thumb area. By default, *XPage* is equal to the current width of the window in *XUnit*s.

See also: **TScroller::SetPageSize**

XPos `long XPos;`

XPos is the scroller's current horizontal position in *XUnit*s.

XRange `long XRange;`

XRange is the total number of horizontal *XUnit*s the window can be scrolled. *XRange* values are supplied to the constructor, but can be modified later by calling **SetRange**.

XUnit `int XUnit;`

XUnit is the horizontal logical scroll unit (expressed in device units) used by the **TScroller**. The *XUnit* value is supplied to the constructor, but can be modified later by calling **SetUnits**.

YLine `int YLine;`

YLine is the number of *YUnit*s to scroll vertically in response to a click on the vertical scroll bar arrow. The default value is 1.

YPage `int YPage;`

YPage is the number of *YUnit*s to scroll vertically in response to a click on the vertical scroll bar's thumb area. By default, *YPage* is equal to the current height of the window in *YUnit*s.

See also: **TScroller::SetPageSize**

YPos `long YPos;`

YPos is the scroller's current vertical position in *YUnit*s.

YRange `long YRange;`

YRange is the total number of vertical *YUnit*s the window can be scrolled. *YRange* values are supplied to the constructor, but can be modified later by calling **SetRange**.

YUnit `int YUnit;`

YUnit is the vertical logical scroll unit (expressed in device units) used by the **TScroller**. The *YUnit* value is supplied to the constructor, but can be modified later by calling **SetUnits**.

Member functions

constructor

```
TScroller(PTWindow TheWindow, int TheXUnit, int TheYUnit,
          long TheXRange, long TheYRange);
```

Constructs a **TScroller** object with *TheWindow* as the owner window, and *TheXUnit*, *TheYUnit*, *TheXRange*, and *TheYRange* as *XUnit*, *YUnit*, *XRange* and *YRange*, respectively. Initializes data members to default values. *HasHScrollBar* and *HasVScrollBar* are set according to the scroll bar attributes of the owner window.

constructor `TScroller(StreamableInit);` **protected**

TScroller stream constructor. Called when a **TScroller** is instantiated using data from an input stream.

destructor `~TScroller();`

Destructs a **TScroller** object. Sets *Scroller* to NULL.

AutoScroll `virtual void AutoScroll();`

Scrolls the owner window's display in response to the mouse being dragged from within the window to without. The direction and the amount by which the display is scrolled depend upon the current position of the mouse.

BeginView `virtual void BeginView(HDC PaintDC, PAINTSTRUCT _FAR & PaintInfo);`

If *AutoOrg* is TRUE (default condition), automatically offsets the origin of the logical coordinates of the client area by XPos, YPos during a paint operation. If *AutoOrg* is FALSE (for example, when the scroller is larger than 32767 units) you must set the offset manually.

build `static PTStreamable build();`

Invokes the **TScroller(StreamableInit)** constructor. Constructs an object of the type **TScroller** prior to reading in its data members from a stream.

EndView `virtual void EndView();`

Updates the position of the owner window's scroll bars to be coordinated with the position of the **TScroller**.

hashValue `virtual hashValueType hashValue() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *InstanceHashValue*, the unique identifier of a **TScroller** instance.

HScroll `virtual void HScroll(WORD ScrollEvent, int ThumbPos);`

Responds to the specified horizontal *ScrollEvent* by calling **ScrollBy** or **ScrollTo**. The type of scroll event is identified by the corresponding Windows SB_ constants (see Chapter 18, "Miscellaneous components"). *ThumbPos* contains the current thumb position when the scroller is notified of SB_THUMBTRACK and SB_THUMBPOSITION scroll events.

See also: **TWindow::WMHScroll**

isA `virtual classType isA() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *scrollerClass*, the class identifier of **TScroller**.

isEqual `virtual int isEqual(RCObject testobj) const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns TRUE if **this** points to the specified *testobj*.

IsVisibleRect `BOOL IsVisibleRect(long X, long Y, int XExt, int YExt);`

Returns TRUE if any portion of the specified rectangle is currently visible in the owner window.

nameOf `virtual Pchar nameOf() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns "TScroller", the class identification string of **TScroller**.

printOn `virtual void printOn(Rostream outputStream) const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Prints a hexadecimal representation of *Window* on the specified stream.

read `virtual Pvoid read(Ripstream is);` **protected**

Reads in *AutoMode*, *AutoOrg*, *HasHScrollBar*, *HasVScrollBar*, *TrackMode*, *XLine*, *XPage*, *XPos*, *XRange*, *XUnit*, *YLine*, *YPage*, *YPos*, *YRange*, and *YUnit*.

See also: **TScroller::write**

ScrollBy `void ScrollBy(long Dx, long Dy);`

Scrolls the owner window's display by *Dx* and *Dy*, by calling **ScrollTo**.

See also: **TScroller::ScrollTo**

ScrollTo `virtual void ScrollTo(long X, long Y);`

Scrolls to the position whose coordinates are supplied, after checking boundary conditions. Causes a **WMPaint** message to be sent. If a portion of the client area will remain visible, first scrolls the contents of the client area.

See also: **TWindow::WMPaint, TWindow::Paint**

SetPageSize `virtual void SetPageSize();`

Sets the *XPage* and *YPage* data members to the width and height (in *XUnit*s and *YUnit*s) of the owner window's client area.

See also: **TWindow::WMSize**

SetRange `void SetRange(long TheXRange, long TheYRange);`

Sets the *XRange* and *YRange* of the **TScroller** to the parameters specified. Then calls **SetSBarRange** to synchronize the range of the owner window's scroll bars.

See also: **TScroller::SetSBarRange**

SetSBarRange `virtual void SetSBarRange();`

Sets the range of the of the owner window's scroll bars match the range of the **TScroller**.

SetUnits `void SetUnits(int TheXUnit, int TheYUnit);`

Sets the *XUnit* and *YUnit* data members to *TheXUnit* and *TheYUnit*, respectively. Updates *XPage* and *YPage* by calling **SetPageSize**.

VScroll `virtual void VScroll(WORD ScrollEvent, int ThumbPos);`

Responds to the specified vertical *ScrollEvent* by calling **ScrollBy** or **ScrollTo**. The type of scroll event is identified by the corresponding

Windows SB_ constants (see the online Help). *ThumbPos* contains the current thumb position when the scroller is notified of SB_THUMBTRACK and SB_THUMBPOSITION scroll events.

See also: **TScroller::ScrollTo**, **TWindow::WMVScroll**

write `virtual void write(Ropstream os);` **protected**

Writes out *AutoMode*, *AutoOrg*, *HasHScrollBar*, *HasVScrollBar*, *TrackMode*, *XLine*, *XPage*, *XPos*, *XRange*, *XUnit*, *YLine*, *YPage*, *YPos*, *YRange*, and *YUnit*.

XRangeValue `long XRangeValue(int AScrollUnit);`

Converts a horizontal scroll value from the scroll bar to a horizontal range value.

XScrollValue `int XScrollValue(long ARangeUnit);`

Converts a horizontal range value from the scroll bar to a horizontal scroll value.

YRangeValue `long YRangeValue(int AScrollUnit);`

Converts a vertical scroll value from the scroll bar to a vertical range value.

YScrollValue `int YScrollValue(long ARangeUnit);`

Converts a vertical range value from the scroll bar to a vertical scroll value.

TSearchDialog

editwnd.h

Retrieves search/replace text and related options from the user. A *TSearchStruct* passed to the **TSearchDialog** constructor transfers data both in and out.

STDWNDS.DLG contains two resource definitions: one for a search dialog box and the other for a replace dialog box. To use a **TSearchDialog**, you must compile and append dialog box resources to your ObjectWindows application.

See also: **TEditWindow::CMEditFind, TEditWindow::CMEditReplace**

Member function

constructor

```
TSearchDialog(PTWindowsObject AParent, int ResourceId, TSearchStruct _FAR
              &SearchStruct, PTModule AModule = NULL);
```

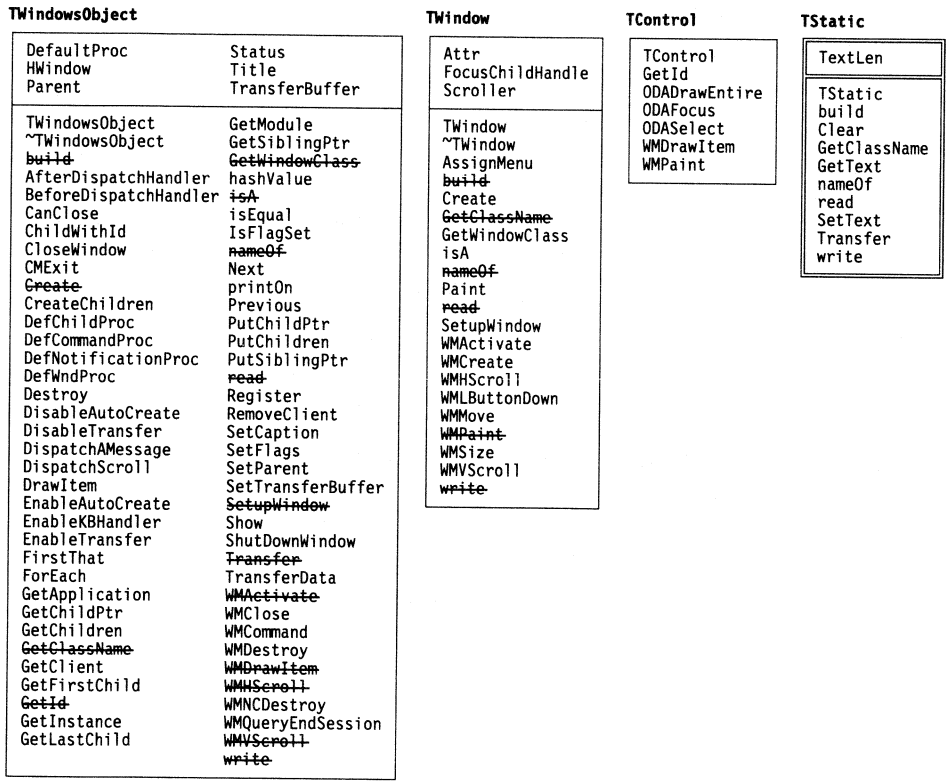
Invokes a **TDialog** constructor, passing a pointer to the specified parent (*AParent*). Constructs **TEdit** and **TCheckBox** control objects to be associated with corresponding search/replace dialog box controls. Sets *TransferBuffer* to the specified search structure. Text and options are transferred between the search structure and the dialog box.

The *ResourceId* parameter should be either SD_SEARCH or SD_REPLACE, corresponding to a dialog template defined in STDWNDS.DLG. If *ResourceId* is SD_SEARCH, a *search* **TSearchDialog** executes. If *ResourceId* is SD_REPLACE, a *replace* **TSearchDialog** executes.

See also: **TWindowsObject::TransferBuffer**

TStatic

static.h



A **TStatic** is an interface object that represents a static text interface element in Windows. A **TStatic** must be used to create a static control in a

parent **TWindow**. A **TStatic** can also be used to facilitate modifications to the text of static controls in **TDialog**s.

Data member

TextLen

```
WORD TextLen;
```

TextLen holds the size of the text buffer for static controls. The number of characters that can actually be stored in the static control is one less than *TextLen* because of the null terminator on the string. *TextLen* is also the number of bytes transferred by the **Transfer** member function.

Member functions

constructor

```
TStatic(PTWindowsObject AParent, int AnId, LPSTR ATitle, int X, int Y,
        int W, int H, WORD ATextLen, PTModule AModule = NULL);
```

Constructs a static control object with the supplied parent window (*AParent*), control ID (*AnId*), text (*ATitle*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), and text length(*ATextLen*). By default, the static control is visible upon creation and has left-justified text. (*Attr.Style* is set to WS_CHILD | WS_VISIBLE | WS_GROUP | SS_LEFT.) Invokes a **TControl** constructor.

See also: **TControl::TControl**

constructor

```
TStatic(PTWindowsObject AParent, int ResourceId, WORD ATextLen,
        PTModule AModule = NULL);
```

Constructs a **TStatic** object to be associated with a static control interface control of a **TDialog**. Invokes the **TControl** constructor with similar parameters, then sets *TextLen* to *ATextLen*. Disables the data transfer mechanism by calling **DisableTransfer**.

The *ResourceId* parameter must correspond to a static control resource that you define.

See also: **TControl::TControl**

constructor

```
TStatic(StreamableInit);
```

protected

TStatic stream constructor. Invokes **TControl**'s stream constructor. Called when a **TStatic** is instantiated using data from an input stream.

See also: **TControl::TControl**

build `static PTStreamable build();`

Invokes the **TStatic(StreamableInit)** constructor. Constructs an object of the type **TStatic** prior to reading in its data members from a stream.

Clear `void Clear();`

Clears the static control's text.

GetClassName `virtual LPSTR GetClassName();` **protected**

Returns the name of **TStatic**'s Windows registration class, "STATIC".

GetText `int GetText(LPSTR ATextString, int MaxChars);`

Retrieves the static control's text and stores it in the *ATextString* argument of *MaxChars* size. Returns the number of characters copied.

GetTextLen `int GetTextLen()`

Returns the length of the static control's text.

nameOf `virtual Pchar nameOf() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns "TStatic", the class identification string of **TStatic**.

read `virtual Pvoid read(Ripstream is);` **protected**

Invokes **TWindow::read** to read in the base **TWindow** object. Then reads in *TextLen*.

SetText `void SetText(LPSTR ATextString);`

Sets the static control's text to the string supplied in *ATextString*.

Transfer `virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);`

Transfers *TextLen* characters of text to or from a transfer buffer pointed to by *DataPtr*. If *TransferFlag* is TF_GETDATA, the text is transferred to the buffer from the static control. If *TransferFlag* is TF_SETDATA, the static control's text is set to the text contained in the transfer buffer. **Transfer** returns *TextLen*, the number of bytes stored in or retrieved from the buffer. If *TransferFlag* is TF_SIZEDATA, *Transfer* simply returns *TextLen* without transferring data.

write `virtual void write(Ropstream os);` **protected**

Invokes **TWindow::write** to write out the base **TWindow** object. Then writes out *TextLen*.

TWindow

window.h

TWindowsObject

DefaultProc	Status
HWindow	Title
Parent	TransferBuffer
TWindowsObject	GetModule
~TWindowsObject	GetSiblingPtr
~~build~~	~~GetWindowClass~~
AfterDispatchHandler	hashValue
BeforeDispatchHandler	~~isA~~
CanClose	isEqual
ChildWithId	IsFlagSet
CloseWindow	~~nameOf~~
CMExit	Next
~~Create~~	printOn
CreateChildren	Previous
DefChildProc	PutChildPtr
DefCommandProc	PutChildren
DefNotificationProc	PutSiblingPtr
DefWndProc	~~read~~
Destroy	Register
DisableAutoCreate	RemoveClient
DisableTransfer	SetCaption
DispatchAMessage	SetFlags
DispatchScroll	SetParent
DrawItem	SetTransferBuffer
EnableAutoCreate	~~SetupWindow~~
EnableKBHandler	Show
EnableTransfer	ShutDownWindow
FirstThat	Transfer
ForEach	TransferData
GetApplication	~~WMActivate~~
GetChildPtr	WMClose
GetChildren	WMCommand
~~GetClassName~~	WMDestroy
GetClient	WMDrawItem
GetFirstChild	~~WMHScroll~~
GetId	WMNCDestroy
GetInstance	WMQueryEndSession
GetLastChild	~~WMVScroll~~
	~~write~~

TWindow

Attr
FocusChildHandle
Scroller
TWindow
~TWindow
AssignMenu
build
Create
GetClassName
GetWindowClass
isA
nameOf
Paint
read
SetupWindow
WMActivate
WMCreate
WMHScroll
WMLButtonDown
WMMove
WMPaint
WMSize
WMVScroll
write

TWindow is derived from **TWindowsObject**, and provides window-specific behavior. (**TDialog** provides behavior unique to a dialog box.) **TWindow** functions specify window creation, including registration and attributes.

TWindow is a generic window (complete with caption) that can be resized and moved. You can construct and create an instance of **TWindow**, though normally you'll use **TWindow** as a base for your specialized window classes.

You can use an instance or derived class of **TWindow** as an MDI child window by simply specifying a **TMDIFrame** as its parent.

Many **TWindow** member functions provide the hooks necessary for the utilization of a **TScroller** object, which aids in scrolling the display of the **TWindow**.

TWindow is also the base class of **TMDIClient**, a specialized control for MDI-compliant applications.

Data members

Attr

```
TWindowAttr Attr;
```

Attr holds a **TWindowAttr** structure that contains the window's creation attributes istics of the window object's corresponding interface element). These attributes include the window's style, extended style, position, size, menu handle, and control ID. See the structure definition for **TWindowAttr** in Chapter 18, "Miscellaneous components."

See also: **TWindow::TWindow, TWindow::Create**

FocusChildHandle

```
HANDLE FocusChildHandle;
```

FocusChildHandle contains the handle of the window's child window that had the focus the last time the window was activated.

Scroller

```
PTScroller Scroller;
```

Scroller, when non-NULL, points to a **TScroller** object used to aid scrolling of the **TWindow**'s display. The **TScroller** object is instantiated, and *Scroller* is set, in the constructor of the **TWindow**.

Member functions

constructor

```
TWindow(PTWindowsObject AParent, LPSTR ATitle, PTModule AModule = NULL);
```

Constructs a window object with the parent window supplied in *AParent*. (*AParent* should be NULL for main windows, which have no parent.) The associated text, usually the window's caption, is copied from *ATitle*.

Invokes a **TWindowsObject** constructor. Sets the position and extent fields in the *Attr* structure to defaults appropriate for overlapped and pop-up windows. If *AParent* is NULL, *Attr.Style* is set to WS_OVERLAPPED-WINDOW. It is set to WS_CLIPSIBLINGS for **TWindow**s, which are children of a **TMDIFrame**. *Attr.Style* is set to WS_VISIBLE for all other **TWindow**s.

The default values in the *Attr* structure can be reset in the constructor of a class derived from **TWindow** (or at any time prior to creation of its interface element). A constructor of a **TWindow** derived class can also set

Scroller (NULL by default) to an instantiated **TScroller** object. Be sure your **TWindow** derived classes' constructors invoke this **TWindow** constructor.

If *AParent* is an MDI frame window, this is created as an MDI child window (the WB_MDICHILD flag is set). If the new **TWindow** is a popup window (and therefore should not be created as a MDI child window) then `SetFlags(WB_MDICHILD, FALSE)` must be called after this is constructed, and before it is created.

See also: **TWindowsObject::TWindowsObject**

constructor

```
TWindow(HWND AnHWindow, PTModule AModule = NULL);
```

Constructor for a **TWindow** that is being used in a DLL as an alias for a non-ObjectWindows window. Queries Windows to initialize data members. Invokes **TWindowsObject** constructor passing *AnHWindow* and *AModule*.

constructor **protected**

```
TWindow(StreamableInit);
```

TWindow stream constructor. Invokes **TWindowsObject**'s stream constructor. Called when a **TWindow** is instantiated using data from an input stream.

See also: **TWindowsObject::TWindowsObject**

destructor

```
virtual ~TWindow();
```

Frees *Attr.Menu* and deletes *Scroller*, if non-NULL.

ActivationResponse

```
virtual void ActivationResponse(WORD Activated,  BOOL IsIconified);
```

Called by **WMActivate** and **WMMDIActivate** to provide a keyboard interface for the controls of a window. First calls **TWindowsObject::ActivationResponse**. When the window is deactivated (*Activated* equals zero), saves the handle of the child control that currently has the focus in *FocusChildHandle*. When the window is activated, sets the focus to the child whose handle was stored.

See also: **TWindowsObject::EnableKBHandler, TWindow::WMMDIActivate**

AssignMenu

```
virtual BOOL AssignMenu(LPSTR MenuName);
```

Sets *Attr.Menu* to the supplied *MenuName*. Frees any previous strings pointed to by *Attr.Menu*. If *HWindow* is nonzero, loads and sets the menu of the window, destroying the previous menu, if any.

AssignMenu

```
virtual BOOL AssignMenu(int MenuId);
```

Assigns the menu ID of the window using the integer resource identifier defined in a menu resource file. Passes the value to the other **AssignMenu** member function, which does the work.

build `static PTStreamable build();`

Invokes the **TWindow(StreamableInit)** constructor. Constructs an object of the type **TWindow** prior to reading in its data members from a stream.

See also: **TWindow::TWindow**

Create `virtual BOOL Create();`

Creates an interface element and associates it with the **TWindow** object. The association is not attempted if *Status* is nonzero. **Create** first calls **Register** to register the window class, if not already registered. The interface element is then created and associated. (If the WB_FROMRESOURCE flag is set, the association is simply made; registration and creation steps are skipped because the interface element has already been created through a resource definition.) **SetupWindow** is then called to set up the newly created window. If successful, **Create** returns TRUE. If not successful, **Error** is called and FALSE is returned.

Normally you will never call **Create** directly. **Create** is called by **TModule::MakeWindow**, which first performs validity checks.

See also: **TWindowsObject::Register**, **TWindowsObject::SetupWindow**, **TModule::MakeWindow**

GetClassName `virtual LPSTR GetClassName();` **protected**

Returns "OWLWindow," the name ofthe default Windows registration class for a **TWindow**.

GetWindowClass `virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);` **protected**

Fills the **WNDCLASS** structure, supplied in *AWndClass*, with the default registration attributes appropriate for a **TWindow**. The *Style* data member is set to CS_HREDRAW or CS_VREDRAW. The icon is set to a generic icon, the cursor is set to the stock arrow cursor, and the background color is set to the system's window background color. The name of the class to be registered is retrieved by calling **GetClassName**.

See also: **TWindow::GetClassName**, **TWindowsObject::Register**, **TWindow::Create**

isA `virtual classType isA() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *windowClass*, the class identifier of **TWindow**.

nameOf `virtual Pchar nameOf() const;`

Required by container classes. Redefines the pure virtual function in class **Object**. Returns "TWindow," the class identification string of **TWindow**.

Paint `virtual void Paint(HDC PaintDC, PAINTSTRUCT _FAR & PaintInfo);` **protected**

Serves as a placeholder for derived types that define **Paint** member functions. **Paint** is called by **WMPaint**, requested automatically by Windows to redisplay the window's contents. Uses *PaintDC* as the paint display context supplied to text and graphics output functions. The supplied reference to the **PaintInfo** structure contains information about the paint operation, including the area that requires painting.

See also: **TWindow::WMPaint**

read `virtual Pvoid read(Ripstream is);` **protected**

Invokes **TWindowsObject::read** to read in the base **TWindowsObject** object. Then, if the WB_FROMRESOURCE flag is not set, reads in *Attr.Style*, *Attr.ExStyle*, *Attr.X*, *Attr.Y*, *Attr.W*, *Attr.H*, and *Attr.Param*. Then reads in *Attr.Id*, *Attr.Menu*, and *Scroller*. It sets *FocusChildHandle* to 0.

SetupWindow `virtual void SetupWindow();` **protected**

Sets up the newly created window. Calls **TWindowsObject::SetupWindow** to create the windows in the child list. If keyboard handling has been requested and *FocusChildHandle* has not been set, sets it to the appropriate child. If the window is an MDI child window, **SetupWindow** calls **SetFocus** to give the new window the focus. If the window has a scroller object, **SetupWindow** calls **SetSBarRange** to set the range of its scroll bars. Can be redefined in derived classes to perform additional initialization. A redefinition in a derived class should invoke **SetupWindow** in its base.

See also: **TScroller::SetSBarRange**

WMCreate
```
virtual void WMCreate(RTMessage Msg)
  = [WM_FIRST + WM_CREATE];
```
protected

Responds to a WM_CREATE message by calling **SetupWindow**.

See also: **TWindow::SetupWindow, TWindow::Create**

WMHScroll
```
virtual void WMHScroll(RTMessage Msg)
  = [WM_FIRST + WM_HSCROLL];
```
protected

Responds to horizontal window scroll bar events. If the message is from a scroll bar control, calls **DispatchScroll**. Otherwise, if *Scroller* is non-NULL,

calls **Scroller->HScroll**. (Assumes, because of a Windows bug, that if it is a window with scroll bar controls, it will not have a scroll bar style.)

See also: **TScroller::HScroll**

WMLButtonDown

```
virtual void WMLButtonDown(RTMessage Msg)
   = [WM_FIRST + WM_LBUTTONDOWN];
```

protected

Response member function for an incoming WM_LBUTTONDOWN message. If the **TWindow** has an auto-scrolling *Scroller*, loops until a WM_LBUTTONUP message comes in, calling **Scroller->AutoScroll**. Captures and releases mouse input prior to and after loop. If you redefine this member function to process mouse clicks, but still want to use auto-scrolling, be sure to call this function from your redefined **WMLButtonDown**.

WMMDIActivate

```
virtual void WMMDIActivate(RTMessage) = [WM_FIRST +
WM_MDIACTIVATE];
```

protected

Calls **DefWndProc**, then responds to an incoming WM_MDIACTIVATE message by setting the parent's (**TMDIFrame**) *ActiveChild* to the window handle sent in the message. Calls **ActivationResponse** to provide a keyboard interface for the controls of a window. If you redefine **WMMDIActivate** in a derived class, be sure to invoke the **WMMDIActivate** of the base class after any other processing (unless you have a special reason for invoking it elsewhere).

See also: **TWindow::ActivationResponse**

WMMove

```
virtual void WMMove(RTMessage Msg) = [WM_FIRST + WM_MOVE];
```

protected

Responds to a WM_MOVE message by saving the coordinates of the window's new position, if not iconic or maximized, in the X and Y members of the *Attr* structure.

WMPaint

```
virtual void WMPaint(RTMessage Msg) = [WM_FIRST + WM_PAINT];
```

protected

Responds to the Windows WM_PAINT message. If the window has a predefined Windows class, calls **DefWndProc**. Otherwise, calls **Paint**, performing Windows-required paint setup and cleanup (**BeginPaint** and **EndPaint**) before and after call. If the window has a *Scroller*, also calls its **BeginView** and **EndView** member functions before and after calling **Paint**.

See also: **TMDIClient::WMPaint, TWindow::Paint, TScroller::BeginView, TScroller::EndView**

WMSize

```
virtual void WMSize(RTMessage Msg) = [WM_FIRST + WM_SIZE];
```

protected

For windows with scrollers, responds to a window sizing event by calling **SetPageSize** to adjust for the new window size. For windows with and

without scrollers, saves the size (if not larger than the display) of the window in *Attr*.

See also: **TScroller::SetPageSize**

WMVScroll

```
virtual void WMVScroll(RTMessage Msg)
  = [WM_FIRST + WM_VSCROLL];
```

protected

If the message is from a scroll bar control, calls **DispatchScroll**. Otherwise, if *Scroller* is non-NULL, calls **Scroller->VScroll**. (Assumes, because of a Windows bug, that if the window has a scroll bar style, it will not have scroll bar controls.)

See also: **TScroller::VScroll**

write

```
virtual void write(Ropstream os);
```

protected

Invokes **TWindowsObject::write** to write out the base **TWindowsObject** object. If the WB_FROMRESOURCE flag is not set, writes out *Attr.Style*, *Attr.ExStyle*, *Attr.X*, *Attr.Y*, *Attr.W*, *Attr.H*, and *Attr.Param*. Then it writes out *Attr.Id*, *Attr.Menu*, and *Scroller*.

TWindowsObject — windobj.h

TWindowsObject is an abstract class, derived from **Object** and **TStreamable**, that defines the fundamental behavior shared by all Object-Windows interface objects (windows, dialog boxes, and controls).

TWindowsObject maintains a list of child windows in its child list and provides behavior that acts upon the child list. For example, when you create a **TWindowsObject**, it creates its child windows. When you attempt to close a **TWindowsObject**, it will close only if its child windows can be closed. **TWindowsObject** additionally defines member functions that provide access to windows in its child list.

TWindowsObject also plays an important role in the mapping of incoming messages to response member functions.

Data members

DefaultProc

```
FARPROC DefaultProc;
```

protected

Holds the address of the default window procedure, which defines the default processing for Windows messages.

HWindow

```
HWND HWindow;
```

HWindow holds a handle to the interface object's associated interface element. If there is no associated element, *HWindow* contains 0. *HWindow* is set to the handle of an associated interface element when it is created and is zeroed when the interface element is destroyed.

Parent

```
PTWindowsObject Parent;
```

Points to the interface object that serves as parent window to this interface object.

Status

```
int Status;
```

Status is used to signal an error in the initialization of an interface object. A nonzero *Status* indicates that the object's initialization was not successful. Classes derived from **TWindowsObject** do not attempt to associate an interface element with an object whose previous initialization has failed. **TWindowsObject**'s derived classes also set *Status* if an attempt to associate an interface element fails.

Possible error values defined by ObjectWindows include EM_INVALIDWINDOW, EM_INVALIDCLIENT, EM_INVALIDCHILD, and EM_INVALIDMAINWINDOW. (See "EM_XXXX constants" in Chapter 18.) You can set *Status* to error values that you define to flag initialization errors. *Status* is set to error values in **Create** or **SetupWindow**.

Title

```
LPSTR Title;
```

Title points to the window's caption.

See also: **TDialog::SetCaption, TDialog::SetupWindow, TWindowsObject::SetCaption**

TransferBuffer

```
Pvoid TransferBuffer;
```

protected

TransferBuffer points to a buffer to be used in transferring data in and out of the **TWindowsObject**. A **TWindowsObject** assumes that the buffer contains data used by the windows in its child list. If NULL, no data is to be transferred.

Member functions

constructor

```
TWindowsObject(PTWindowsObject AParent, PTModule AModule = NULL);
```

Sets *Parent* to *AParent*. Adds **this** to the child list of *AParent*, if non-NULL, and calls **EnableAutoCreate** so that **this** will be created and displayed along with *AParent*.

When passing **TWindowsObject** descendants into DLLs (explicitly or implicitly), it is best to pass in *AModule* explicitly.

AModule specifies the application or DLL module that owns the **TWindowsObject** instance. ObjectWindows needs the correct value of *AModule* to find needed resources. If *AModule* is zero (its default value), **TWindowsObject** sets its module (retrievable with **GetModule**) according to the following rules:

- If the **TWindowsObject** constructor is invoked from an application, the module is set to the application.
- If the **TWindowsObject** constructor is invoked from a DLL that is dynamically linked with the ObjectWindows DLL (see the online README file for directions to programming information on using ObjectWindows as a DLL) and the currently running application is linked the same way, the module is set to the currently running application.
- If the **TWindowsObject** constructor is invoked from a DLL that is statically linked with the ObjectWindows library (for example OWLWL.LIB) or the invoking DLL is dynamically linked with ObjectWindows DLL but the currently running application is not, no default is used for setting the module. Instead, *Status* is set to EM_INVALIDMODULE and creation of the object fails.

See also: **TWindowsObject::EnableAutoCreate**

constructor `TWindowsObject(StreamableInit);` **protected**

TWindowsObject stream constructor. Called when a derived class is instantiated using data from an input stream.

destructor `virtual ~TWindowsObject();`

Destroys a still-associated interface element by calling **Destroy**. Deletes the window objects in its child list, then removes **this** from *Parent*'s child list.

build `static PTStreamable build();`

Invokes the **TWindowsObject(StreamableInit)** constructor. Constructs an object of the type **TWindowsObject** prior to reading in its data members from a stream.

ActivationResponse `virtual void ActivationResponse(WORD Activated, BOOL IsIconified);`

Called by **TWindowsObject::WMActivate** and **TWindow::ActivationResponse**. If this is being activated (*Activated* is non-zero) and keyboard handling has been requested for this, enables

keyboard handling by calling **GetApplication()->SetKBHandler**; otherwise disables it. *IsIconified* is TRUE if the window is minimized, FALSE otherwise.

See also: **TWindowsObject::WMActivate, TWindowsObject::EnableKBHandler, TApplication::SetKBHandler, TWindow::ActivationResponse**

fterDispatchHandler

```
virtual void AfterDispatchHandler();
```

Redefine to perform postprocessing for an incoming message (otherwise, does nothing by default). Called by **DispatchAMessage** after invoking a message response.

See also: **TWindowsObject::DispatchAMessage**

foreDispatchHandler

```
virtual void BeforeDispatchHandler();
```

Redefine to perform preprocessing for an incoming message (otherwise, does nothing by default). Called by **DispatchAMessage** before invoking a message response.

See also: **TWindowsObject::DispatchAMessage**

CanClose

```
virtual BOOL CanClose();
```

Returns TRUE if the associated interface element can be closed. Calls the **CanClose** member function of each of its child windows; returns FALSE if FALSE is returned from any of the **CanClose** calls.

ChildWithId

```
PTWindowsObject ChildWithId(int Id);
```

Returns a pointer to the window in the child window list with the supplied ID. Returns NULL if no child window has the specified *Id*.

CloseWindow

```
void CloseWindow();
```

Calls **ShutDownWindow** (after determining that it's OK to do so). If **this** is the main window of the application, calls **GetApplication()->CanClose**. Otherwise, calls **this->CanClose** to determine whether to window can be closed.

See also: **TWindowsObject::CanClose, TApplication::CanClose, ShutDownWindow, WMClose**

CMExit

```
virtual void CMExit(RTMessage Msg)
  = [CM_FIRST + CM_EXIT];
```

protected

Called in response to the selection of a menu item with an ID of CM_EXIT. If **this** is the main window, calls **CloseWindow**.

Create

```
virtual BOOL Create() = 0;
```

A pure virtual function redefined in derived types to create the Object-Windows interface objects associated Windows interface element.

CreateChildren

```
BOOL CreateChildren();
```

Creates the child windows in the child list whose auto-create flags (with WB_AUTOCREATE mask) are set.

See also: **TWindowsObject::EnableAutoCreate**, **TWindowsObject::DisableAutoCreate**

DefChildProc

```
virtual void DefChildProc(RTMessage Msg);
```

protected

Performs default processing for an incoming child-ID-based message. Calls **DefWndProc**, by default.

This member function can be redefined to respond to messages from any child window for which no corresponding child-ID-based response has been defined. Responding to messages from child windows in this way is at times more convenient than defining a child-ID-based response member function for each child.

DefCommandProc

```
virtual void DefCommandProc(RTMessage Msg);
```

protected

Performs default processing for an incoming command-based message. Looks for the original message receiver, if this window, calls **DefWndProc**; if a parent window, gives the message to the parent; otherwise to the original receiver.

This member function can be redefined to respond to command messages for which no corresponding command-based response member has been defined. Responding to command-based messages in this way is at times more convenient than defining a command-based response member function for each command.

DefNotificationProc

```
virtual void DefNotificationProc(RTMessage Msg);
```

protected

Performs default processing for an incoming notification message generated by **this**, passing the message to the parent as a child-ID-based message. (**this** has the option to perform processing in response to its own notification messages.)

This member function could be redefined to respond to all notification messages sent by **this**. Responding in this way is at times more convenient than defining a notify-based response member function for each notification message.

DefWndProc

```
virtual void DefWndProc(RTMessage Msg);
```

Specifies default processing for an incoming message. Invokes default processing defined by Windows. Calls the default window procedure of a window and sets *Msg.Result* to the returned value. Sets *Msg.Result* to 0 if **this** is a dialog box, so that Windows will invoke default processing.

Destroy

```
virtual void Destroy();
```

Destroys an associated interface element after calling **EnableAutoCreate** for each window in the child list. This ensures that windows in the child list will be re-created if **this** is re-created.

See also: **TWindowsObject::WMDestroy**, **TWindowsObject::EnableAutoCreate**

DisableAutoCreate

```
void DisableAutoCreate();
```

Disables the feature that allows an associated child window interface element to be created and displayed along with its parent window. Call **DisableAutoCreate** for pop-up windows and controls if you want to create and display them at a time later than their parent windows.

See also: **TWindowsObject::EnableAutoCreate**.

DisableTransfer

```
void DisableTransfer();
```

Disables, for the interface object, the transfer mechanism, which allows state data to be transferred to and from a transfer buffer.

ispatchAMessage

```
virtual void DispatchAMessage(WORD AMsg, RTMessage AMessage,
                              void (TWindowsObject::* _FAR) (RTMessage));
```

DispatchAMessage dispatches an incoming Windows message to the proper response member function. If a response is not defined, calls the supplied default member function instead. Can be useful for setting breakpoints while debugging.

Calls **BeforeDispatchHandler** and **AfterDispatchHandler**, before and after dispatching the message.

DispatchScroll

```
void DispatchScroll(RTMessage Msg);
```

protected

Called by **WMHScroll** and **WMVScroll** to dispatch messages from scroll bar controls.

See also: **TWindowsObject::WMHScroll**, **TWindowsObject::WMVScroll**

DrawItem

```
virtual void DrawItem(DRAWITEMSTRUCT _FAR & DrawInfo);
```

Redefine to specify the manner in which a drawable child control or drawable menu item is to be drawn. Called when a control (which doesn't know how to draw itself) or menu item must be drawn. Also called when

the selection or focus is shifted to or from the control or item so that its appearance can be modified to reflect its state. A drawable control can also draw itself by redefining the appropriate functions.

EnableAutoCreate `void EnableAutoCreate();`

Ensures that an associated child window interface element is created and displayed along with its parent window. This feature is enabled, by default, for windows and controls, but disabled for dialog boxes. Call **EnableAutoCreate** if you want to create and display a dialog box along with its parent window.

See also: **TWindowsObject::DisableAutoCreate**

EnableKBHandler `void EnableKBHandler();`

Enables a keyboard interface for the controls of a window or a modeless dialog box, allowing the user to use the tab and arrow keys to move between the controls. By default, the keyboard interface is disabled for windows and dialog boxes. (This feature is provided automatically by Windows for dialog boxes.)

See also: **TWindowsObject::WMActivate, TApplication::SetKBHandler**

EnableTransfer `void EnableTransfer();`

Enables the transfer mechanism, which allows state data to be transferred between the window and a transfer buffer.

FirstThat `PTWindowsObject FirstThat(TCondFunc Test, Pvoid PParamList);`

Iterates over the child list calling a Boolean *Test* function, passing each child window in turn as an argument (along with *PParamList*). If a *Test* call returns TRUE, the iteration is stopped and **FirstThat** returns the child window object that was supplied to *Test*. Otherwise, **FirstThat** returns NULL. For information on **TCondFunc**, see Chapter 18, "Miscellaneous components."

In the following example, **GetFirstChecked** calls **FirstThat** to obtain a pointer (*P*) to the first check box in the child list that is checked.

```
BOOL IsThisBoxChecked(Pvoid P, Pvoid) {
   return ((PTCheckBox)P->GetCheck() == BF_CHECKED);
}

PTCheckBox TMyWindow::GetFirstChecked(); {
   return (FirstThat(IsThisBoxChecked, NULL))
}
```

FirstThat `PTWindowsObject FirstThat(TCondMemFunc Test, Pvoid PParamList);`

Refer to the previous description of **FirstThat**. The difference between the two **FirstThat**s is that the previous one takes a function and this one takes a member function as a parameter. See Chapter 18, "Miscellaneous components" for information on **TCondMemFunc**.

ForEach `void ForEach(TActionFunc Action, Pvoid PParamList);`

Iterates over the child list calling a function supplied as the *Action* to be performed, passing each child window in turn as an argument (along with *PParamList*). For information on **TActionFunc**, see Chapter 18, "Miscellaneous components."

In the following example, **CheckAllBoxes** calls **ForEach**, checking all the check boxes in the child list.

```
void CheckTheBox(Pvoid P, Pvoid) {
   (PTCheckBox)P->Check();
}

void CheckAllBoxes(); {
   ForEach(CheckTheBox, NULL);
}
```

ForEach `void ForEach (TActionMemFunc Action, Pvoid PParamList);`

Refer to the previous **ForEach** description. The difference between the two **ForEach** members is that the first takes a function but this one takes a member function as a parameter. See Chapter 18, "Miscellaneous components" for information on **TActionMemFunc**.

GetApplication `PTApplication GetApplication();`

Gets a pointer to the **TApplication** object associated with **this**.

GetChildPtr `void GetChildPtr(Ripstream is, RPTWindowsObject P);` **protected**

Reads a reference to a pointer to a child window from the specified stream. **GetChildPtr** should be called to read a child pointer written by **PutChildPtr**.

See also: **TWindowsObject::GetSiblingPtr**, **TWindowsObject::PutSiblingPtr**, **TWindowsObject::PutChildPtr**

GetChildren `void GetChildren(Ripstream is);`

Reads child windows from the supplied stream into child list. **GetChildren** should only be called during a **read** operation to read child windows that were written to the stream by **PutChildren**.

This member function assumes that the child list is currently empty!

GetClassName

```
virtual LPSTR GetClassName() = 0;
```

protected

Pure virtual function which a derived class must redefine. Returns the Windows registration class name.

See also: **TWindowsObject::GetWindowClass**

GetClient

```
virtual PTMDIClient GetClient();
```

Returns NULL for all non-MDI interface objects that have no MDI client windows. **TMDIFrame** redefines this member function to return a pointer to its MDI client window.

GetFirstChild

```
PTWindowsObject GetFirstChild();
```

Returns a pointer to the first child window in the interface object's child list.

GetId

```
virtual int GetId();
```

Returns the Id used to find the window in a specified parent's child list. **GetId** is redefined by **TControl** descendants to return the object's resource identifier (*Attr.Id*).

By default, **TWindowsObject::GetId** returns 0. This hides any non-control window with a 0 Id. If you need to address individual windows, redefine **GetId** to return a nonzero number.

See also: **TControl::GetId**

GetInstance

```
FARPROC GetInstance();
```

Returns the instance thunk of the window.

GetLastChild

```
PTWindowsObject GetLastChild();
```

Returns a pointer to the last child window in the interface object's child list.

GetModule

```
PTModule GetModule();
```

Returns a pointer to the **TModule** that owns this object.

GetSiblingPtr

```
void GetSiblingPtr(Ipstream is, RPTWindowsObject P);
```

protected

Reads a reference to a pointer to a sibling window from the supplied stream. A sibling window is a window with the same parent as this window—a **TCheckBox**'s **TGroupBox**, for example, is a sibling of the **TCheckBox** in a dialog box. **GetSiblingPtr** should only be used during a **read** operation on a child pointer written by a call to **PutSiblingPtr**. The

value loaded into *P* does not become valid until the window's parent completes its **read** operation; therefore, dereferencing a sibling window pointer within a stream constructor does not produce the correct result.

See also: **TWindowsObject::PutSiblingPtr, TWindowsObject::GetChildPtr, TWindowsObject::PutChildPtr**

GetWindowClass

```
virtual void GetWindowClass(WNDCLASS _FAR &AWndClass);
```

protected

Redefined by derived classes to fill the supplied Windows registration class structure with registration attributes.

hashValue

```
virtual hashValueType hashValue() const;
```

Required by container classes. Redefines the pure virtual function in class **Object**. Returns *HWindow*, the hash value of a **TWindowsObject**.

isA

```
virtual classType isA() const = 0;
```

Pure virtual function. Redefine in derived classes to return a unique class identifier.

isEqual

```
virtual int isEqual(RCObject testwin) const;
```

Required by container classes. Redefines the pure virtual function in class **Object**. Returns TRUE if **this** points to the specified *testwin*.

IsFlagSet

```
BOOL IsFlagSet(WORD Mask);
```

Returns the state of the bit flag whose *Mask* is supplied. Returns TRUE if the bit flag is set, and FALSE if it is not set.

See also: **TWindows Object::SetFlags**

nameOf

```
virtual Pchar nameOf() const = 0;
```

Pure virtual function. Redefine in derived classes to return a class identification string.

Next

```
PTWindowsObject Next();
```

Returns a pointer to the next window in the parent window's child list. If this was the last child added to the list, returns a pointer to the first child added.

See also: **TWindowsObject::Previous**

printOn

```
virtual void printOn(Rostream outputStream) const;
```

Required by container classes. Redefines the pure virtual function in class **Object**. Prints a hexadecimal representation of *HWindow* on the specified stream.

Previous `PTWindowsObject Previous();`

Returns a pointer to the previous window in the parent window's child list.

See also: **TWindowsObject::Next**

PutChildPtr `void PutChildPtr(Ropstream os, PTWindowsObject P);` **protected**

Writes a child window to the supplied stream. **PutChildPtr** should only be called during a **write** operation to write child pointers that can later be read by **GetChildPtr**.

See also: **TWindowsObject::GetSiblingPtr, TWindowsObject::PutSiblingPtr, TWindowsObject::GetChildPtr**

PutChildren `void PutChildren(Ropstream os);`

Writes child windows in the child list to the supplied stream. **PutChildren** should only be called during a **write** operation to write child windows that can later be read by **GetChildren**.

PutSiblingPtr `void PutSiblingPtr(Ropstream os, PTWindowsObject P);` **protected**

Writes a pointer to a sibling window to the specified stream. A sibling window is a window with the same parent as this window. **PutSiblingPtr** should only be used during a **write** operation to write sibling pointers that can later be read by **GetSiblingPtr**.

See also: **TWindowsObject::GetSiblingPtr, TWindowsObject::GetChildPtr, TWindows Object::PutChildPtr**

read `virtual Pvoid read(Ripstream is);` **protected**

Sets child list, *HWindow*, *Parent*, and *TransferBuffer* to 0. Creates an object instance. Reads in *Title*, *Status*, flags, and *CreateOrder*. Calls **GetChildren** to read in the child windows.

Register `virtual BOOL Register();`

Registers the Windows registration class of **this**, if not already registered. Calls **GetClassName** and **GetWindowClass** to retrieve the Windows registration class name and attributes of **this**. **Register** returns TRUE if **this** is registered.

See also: **TWindowsObject::GetClassName, TWindowsObject::GetWindowClass**

RemoveClient

```
void RemoveClient();
```

protected

Removes the specified client window from its child list. The client window must be processed separately from its siblings.

SetCaption

```
void SetCaption(LPSTR ATitle);
```

Copies *ATitle* to an allocated string pointed to by *Title*. Sets the caption of the interface element to *ATitle*. Deletes any previous title.

SetFlags

```
void SetFlags(WORD Mask, BOOL OnOff);
```

Turns on or off a bit flag, depending on the Boolean parameter supplied in *OnOff*. If TRUE is supplied, the bits in *Mask* are set. Otherwise, the bit is cleared. *Mask* can be any one, or a combination, of the WB_ constants.

See also: **TWindowsObject::IsFlagSet**

SetParent

```
virtual void SetParent(PTWindowsObject NewParent);
```

Sets *Parent* to the specified parent window object. Removes **this** from the child list of the previous parent window, if any, and adds **this** to the new parent's child list.

SetTransferBuffer

```
void SetTransferBuffer (Pvoid ATransferBuffer);
```

Sets *TransferBuffer* to *ATransferBuffer*.

SetupWindow

```
virtual void SetupWindow();
```

protected

Performs setup following creation of an associated interface element. Iterates though the child list, attempting to create an associated interface element for each child window object for whom auto-creation is enabled. (By default, auto-creation is enabled for windows and controls, and disabled for dialog boxes.) If a child window cannot be created, *Status* is set to EM_INVALIDCHILD. **SetupWindow** then calls **TransferData**. Can be redefined in derived classes to perform additional special initialization.

Show

```
void Show(int ShowCmd);
```

Show displays the interface element on the screen in a manner specified by the value supplied in *ShowCmd*. The allowable values for *ShowCmd* are shown in the following table:

Value	Description
SW_HIDE	Hidden
SW_SHOW	In the window's current size and position
SW_SHOWMAXIMIZED	Maximized and active
SW_SHOWMINIMIZED	Minimized and active
SW_SHOWNORMAL	Restored and active

These values are defined in windows.h.

ShutDownWindow

```
virtual void ShutDownWindow();
```

Unconditionally destroys the associated interface element and deletes **this**.

Transfer

```
virtual WORD Transfer(Pvoid DataPtr, WORD TransferFlag);
```

Transfers window data to and from the supplied data buffer. The *TransferFlag* supplied specifies whether data is to be read from or written to the supplied buffer, or whether the size of the transfer data is simply to be returned. The return value is the size (in bytes) of the transfer data. This member function simply returns 0 and is redefined in **TControl** derived classes.

TransferData

```
virtual void TransferData(WORD Direction);
```

If *TransferBuffer* is not NULL, transfers data to or from the buffer and the interface object's participating child windows. (Data is not transfered between any child windows whose WB_TRANSFER flag is not set.) **TransferData** calls the **Transfer** member function of each participating child window, passing a pointer to the transfer data as well as the direction specified in *Direction* (TF_SETDATA, TF_GETDATA, or TF_SIZEDATA).

See also: **TWindowsObject::EnableTransfer, TWindowsObject::DisableTransfer, TWindowsObject::SetupWindow**

WMActivate

```
virtual void WMActivate(RTMessage Msg) = [WM_FIRST +
       WM_ACTIVATE];
```

protected

Responds to an incoming WM_ACTIVATE message. Enables or disables keyboard handling by calling **ActivationResponse**.

See also: **TApplication::SetKBHandler, TWindowsObject::EnableKBHandler**

WMClose

```
virtual void WMClose(RTMessage Msg)
   = [WM_FIRST + WM_CLOSE];
```

protected

Responds to a request to close the window by calling **CloseWindow**.

See also: **TWindowsObject::Destroy**

WMCommand

```
virtual void WMCommand(RTMessage Msg)
  = [WM_FIRST + WM_COMMAND];
```

protected

Dispatches command-based, child-ID-based, and notify-based messages, calling the appropriate response member functions.

WMDestroy

```
virtual void WMDestroy(RTMessage Msg)
  = [WM_FIRST + WM_DESTROY];
```

protected

Responds to an incoming WM_DESTROY message. If **this** is the main window, posts a quit message to end the application.

WMDrawItem

```
virtual void WMDrawItem(RTMessage Msg)
  = [WM_FIRST + WM_DRAWITEM];
```

protected

Dispatches WM_DRAWITEM messages for drawable controls by calling the control's **WMDrawItem** member function. Calls **WMDrawItem** if the WM_DRAWITEM message is for a drawable menu item.

See also: **TWindowsObject::DrawItem**, **TControl::WMDrawItem**

WMHScroll

```
virtual void WMHScroll(RTMessage Msg)
  = [WM_FIRST + WM_HSCROLL];
```

protected

Responds to an incoming WM_HSCROLL message by calling **DispatchScroll**.

See also: **TWindowsObject::DispatchScroll**

WMNCDestroy

```
virtual void WMNCDestroy(RTMessage Msg)
  = [WM_FIRST + WM_NCDESTROY];
```

protected

Responds to an incoming WM_NCDESTROY message, the last message sent to a Windows interface element. Zeroes *HWindow* to indicate that an interface element is no longer associated with **this**. Deletes **this** if it is an aliased window object.

MQueryEndSession

```
virtual void WMQueryEndSession(RTMessage Msg)
  = [WM_FIRST + WM_QUERYENDSESSION];
```

protected

Responds to Windows attempt to close down. If **this** is the main window, returns the result of call to **GetApplication()->CanClose**; otherwise, returns the result of call to **this->CanClose**.

WMVScroll `virtual void WMVScroll(RTMessage Msg)`
`= [WM_FIRST + WM_VSCROLL];` **protected**

Responds to an incoming WM_VSCROLL message by calling **DispatchScroll**.

See also: **TWindowsObject::DispatchScroll**

write `virtual void write(Ropstream os);` **protected**

Writes out *Title*, *Status*, flags, and *CreateOrder*. Calls **PutChildren** to write out child windows.

C H A P T E R

17

Streamable class reference

This chapter alphabetically lists all the public classes that support streamable objects. It explains their use, members, operators, and friends.

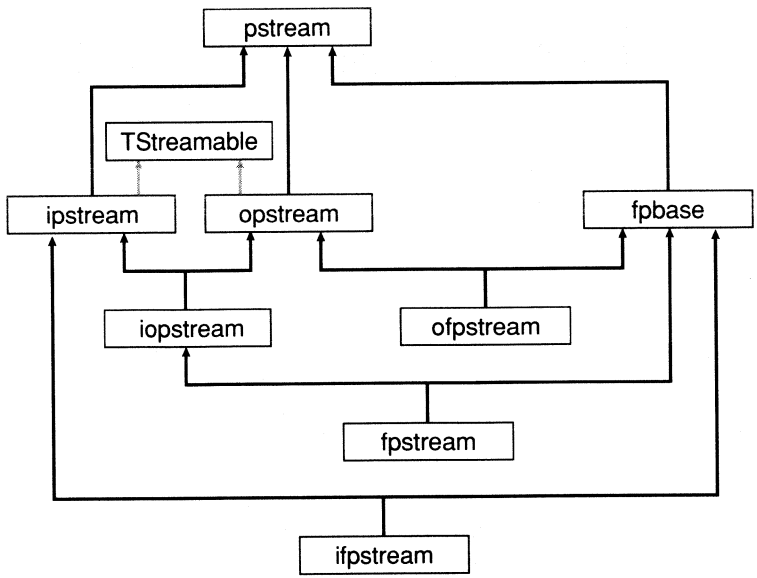
Finding information on a specific class is explained in Chapter 16 on page 223. Refer to the sample class entry format on page 225.

See online Help for more detailed information on streamable class member functions with more than one form.

The streams hierarchy

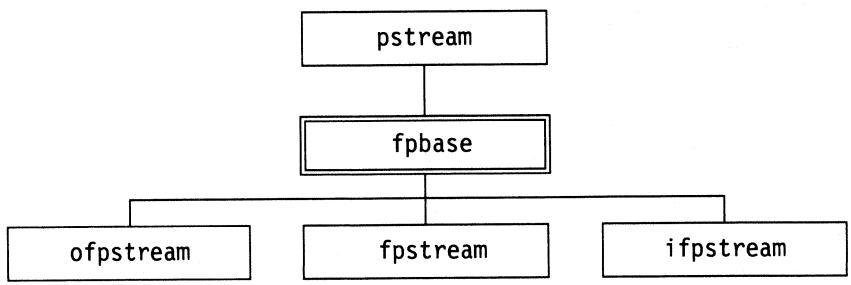
ObjectWindows uses a library consisting of a hierarchy of classes related to streamable objects. This chapter is a reference to these classes as shown in the following figure.

Figure 17.1
Streamable class hierarchy used by ObjectWindows



☞ The gray arrows connecting **TStreamable** show that it is a friend class.

fpbase objstm.h



fpbase provides the basic operations common to all object file stream I/O.

Member functions

constructor

```
fpbase();
fpbase(PCchar name, int omode, int prot = filebuf::openprot);
fpbase(int f);
fpbase(int f, Pchar b, int len);
```

Creates a buffered **fpbase** object. You can set the size and initial contents of the buffer with the *len* and *b* arguments. You can open a file and attach it to the stream by specifying the name, mode, and protection (*prot*) arguments, or by using the file descriptor, *f*.

destructor

```
~fpbase();
```

Destroys the **fpbase** object.

attach

```
void attach(int f);
```

Attaches the file with descriptor *f* to this stream if possible. Sets **ios::state** accordingly.

close

```
void close();
```

Closes the stream and associated file.

open

```
void open(PCchar name, int mode, int prot = filebuf::openprot);
```

Opens the the named file in the given *mode* (*app, ate, in, out, binary, trunc, nocreate, noreplace*) and protection. The opened file is attached to this stream.

rdbuf

```
Pfilebuf rdbuf();
```

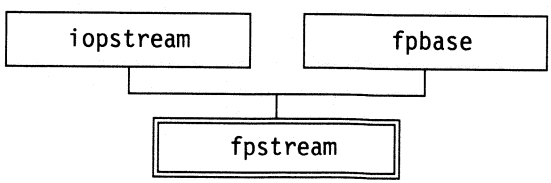
Returns a pointer to the current file buffer.

setbuf

```
void setbuf(Pchar buf, int len);
```

Allocates a buffer of size *len*.

fpstream — objstrm.h



fpstream is a simple "mix" of its bases, **fpbase** and **iopstream**. It provides the base class for simultaneous writing and reading streamable objects to bidirectional file streams.

Member functions

constructor

```
fpstream();
fpstream(PCchar name, int mode, int prot = filebuf::openprot);
fpstream(int f);
fpstream(int f, Pchar b, int len);
```

Creates a buffered **fpstream** object. You can set the size and initial contents of the buffer with the *len* and *b* arguments. You can open a file and attach it to the stream by specifying the name, mode, and protection arguments, or by using the file descriptor, *f*.

destructor

```
~fpstream();
```

Destroys the **fpstream** object.

open

```
void open(PCchar name, int mode,int = prot filebuf::openprot);
```

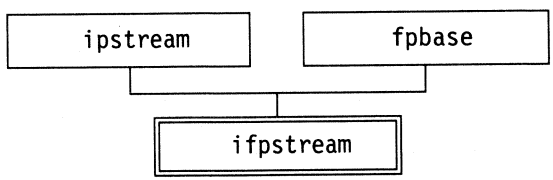
Opens the named file in the given *mode* (*app, ate, in, out, binary, trunc, nocreate, noreplace*) and protection. The opened file is attached to this stream.

rdbuf

```
Pfilebuf rdbuf();
```

Returns the data member *bp*.

ifpstream objstrm.h



ifpstream is a simple "mix" of its bases, **fpbase** and **ipstream**. It provides the base class for reading (extracting) streamable objects from file streams.

Member functions

constructor

```
ifpstream();
ifpstream(PCchar name, int mode = ios::in, int prot = filebuf::openprot);
ifpstream(int f);
ifpstream(int f, Pchar b, int len);
```

Creates a buffered **ifpstream** object. You can set the size and initial contents of the buffer with the *len* and *b* arguments. You can open a file and attach it to the stream by specifying the name, mode, and protection arguments, or via the file descriptor, *f*.

destructor

```
~ifpstream();
```

Destroys the **ifpstream** object.

open

```
void open(PCchar name, int mode = ios::in, int prot = filebuf::openprot);
```

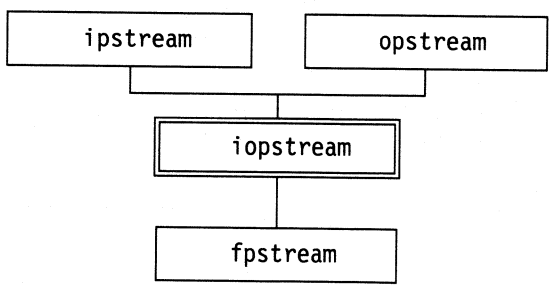
Opens the the named file in the given *mode* (*app, ate, in, out, binary, trunc, nocreate,* or *noreplace*) and protection. The default mode is *in* (input) with *openprot* protection. The opened file is attached to this stream.

rdbuf

```
Pfilebuf rdbuf();
```

Returns a pointer to the current file buffer.

iopstream — objstrm.h



iopstream is a simple "mix" of its bases, **opstream** and **ipstream**. It provides the base class for simultaneous writing and reading of streamable objects.

Member functions

constructor

```
iopstream(Pstreambuf buf);
iopstream();
```

protected

The first form creates a buffered **iopstream** with the given buffer and sets the *bp* data member to *buf*. The state is set to 0. The second form of the constructor does not initialize the buffer pointer *bp*. Use **init** to set the buffer and state.

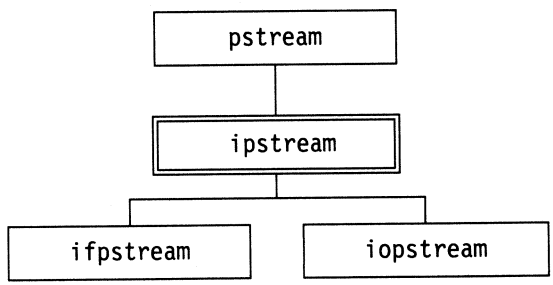
destructor

```
~iopstream();
```

Destroys the **iopstream** object.

See also: **pstream::init**

ipstream objstrm.h



ipstream, a specialized input stream derivative of **pstream**, is the base class for reading (extracting) streamable objects.

Member functions

constructor

```
ipstream(Pstreambuf );
ipstream();
```

protected

Creates a buffered **ipstream** with the given buffer and sets the *bp* data member to *buf*. The state is set to 0. The second form of the constructor does not initialize the buffer pointer *bp*. Use **init** to set the buffer and state.

destructor `~ipstream();`

Destroys the **ipstream** object.

See also: **pstream::init**

find `PCvoid find(P_id_type Id);` **protected**

Returns a pointer to the object corresponding to *Id*.

freadBytes `void freadBytes( void far *data, size_t sz );`

Reads into the supplied far buffer (*data*) the number of bytes specified by *sz*.

freadString `char far *freadString();`

Reads a string from the stream. Determines the length of the string and allocates a far character array of the appropriate length. Reads the string into this array and returns a pointer to the string. The caller is expected to free the allocated memory block.

freadString `char far *freadString( char far *buf, unsigned maxLen );`

Reads a string from the stream into the supplied far buffer (*buf*). If the length of the string is greater than *maxLen*-1, reads nothing. Otherwise reads the string into the buffer and appends a null terminating byte.

readByte `uchar readByte();`

Returns the character at the current stream position.

readBytes `void readBytes(Pvoid data, size_t sz);`

Reads *sz* bytes from current stream position, and writes them to *data*.

readData `Pvoid readData(PCTStreamableClass, PTStreamable mem);` **protected**

Invokes the appropriate **read** function to read from the stream to the object pointed to by *mem*. If *mem* is 0, the appropriate **build** function is called first.

See also: **TStreamableClass**, and the **read** and **build** member functions of each streamable class

readPrefix `PCTStreamableClass readPrefix();` **protected**

Returns the **TStreamableClass** object corresponding to the class **name** stored at the current position.

readString
```
Pchar readString();
Pchar readString(Pchar buf, unsigned maxLen);
```

readString() allocates a buffer large enough to contain the string at the current stream position. Reads the string from the stream into the buffer. The caller must free the buffer.

readString(Pchar buf, unsigned maxLen) reads the string at the current stream position into the buffer specified by *buf*. Does not read more than *maxLen* bytes.

readSuffix `void readSuffix();` **protected**

Reads and checks the final byte of an object's name field.

See also: **ipstream::readPrefix**

readWord
```
ushort readWord();
```

Returns the word at the current stream position.

registerObject `void registerObject(PCvoid adr);` **protected**

Registers the class of the object pointed to by *adr*.

seekg
```
Ripstream seekg(streampos pos);
Ripstream seekg(streamoff off, seek_dir dir);
```

The first form moves the stream position to the absolute position given by *pos*. The second form moves to a position relative to the current position by an offset *off* (+ or –) starting at *dir*. *dir* can be set to *beg* (start of stream), *cur* (current stream position), or *end* (end of stream).

tellg
```
streampos tellg();
```

Returns the (absolute) current stream position.

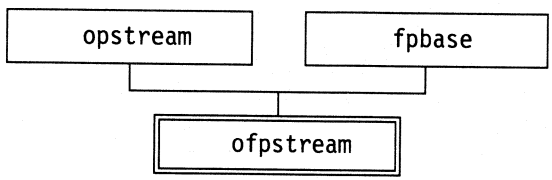
Friends

operator >>
```
friend Ripstream operator >> (Ripstream ps, signed char _FAR & ch);
friend Ripstream operator >> (Ripstream ps, unsigned char _FAR & ch);
friend Ripstream operator >> (Ripstream ps, signed short _FAR & sh);
friend Ripstream operator >> (Ripstream ps, unsigned short _FAR & sh);
friend Ripstream operator >> (Ripstream ps, signed int _FAR & i);
friend Ripstream operator >> (Ripstream ps, unsigned int _FAR & i);
friend Ripstream operator >> (Ripstream ps, signed long _FAR & l);
friend Ripstream operator >> (Ripstream ps, unsigned long _FAR & l);
friend Ripstream operator >> (Ripstream ps, float _FAR & f);
friend Ripstream operator >> (Ripstream ps, double _FAR & d);
friend Ripstream operator >> (Ripstream ps, long double _FAR & d);
friend Ripstream operator >> (Ripstream ps, RTStreamable t);
```

```
friend Ripstream operator >> (Ripstream ps, RPvoid t);
```

Extracts (reads) from the **ipstream** *ps*, to the given argument. A reference to the stream is returned, letting you chain **>>** operations in the usual way. The data type of the argument determines how the read is performed. For example, reading a signed **char** is implemented using **readByte**.

ofpstream — objstrm.h



ofpstream is a simple "mix" of its bases, **fpbase** and **opstream**. It provides the base class for writing (inserting) streamable objects to file streams.

Member functions

constructor

```
ofpstream();
ofpstream(PCchar name, int mode = ios::out,
          int prot = filebuf::openprot);
ofpstream(int f);
ofpstream(int f, Pchar b, int len);
```

Creates a buffered **ofpstream** object. You can set the size and initial contents of the buffer with the *len* and *b* arguments. A file can be opened and attached to the stream by specifying the name, mode, and protection arguments, or by using the file descriptor, *f*.

destructor

```
~ofpstream();
```

Destroys the **ofpstream** object.

open

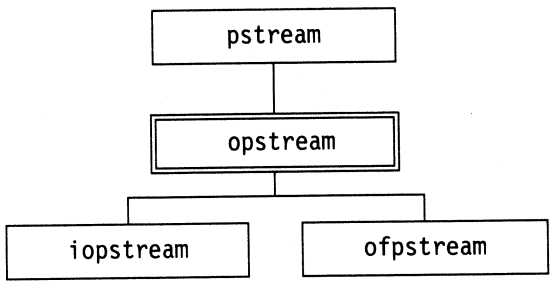
```
void open(PCchar name, int mode = ios::out,
          int prot = filebuf::openprot);
```

Opens the the named file in the given *mode* (*app, ate, in, out, binary, trunc, nocreate,* or *noreplace*) and protection. The default mode is *out* (output) with *openprot* protection. The opened file is attached to this stream.

rdbuf `Pfilebuf rdbuf();`

Returns the current file buffer.

opstream — objstrm.h



opstream, a specialized derivative of **pstream**, is the base class for writing (inserting) streamable objects.

Member functions

constructor
```
opstream(Pstreambuf buf);
opstream();
```
protected

The first form creates a buffered **opstream** with the given buffer and sets the *bp* data member to *buf*. The state is set to 0. The second form allocates a default buffer. The second form of the constructor does not initialize the buffer pointer *bp*. Use **init** to set the buffer and state.

destructor `~opstream();`

Destroys the **opstream** object.

See also: **pstream::init**

find `P_id_type find(PCvoid adr);` **protected**

Returns the type ID for the object pointed to by *adr*.

flush `Ropstream flush();`

Flushes the stream.

fwriteBytes

```
void fwriteBytes( const void far *data, size_t sz );
```

Writes the specified number of bytes (*sz*) from the supplied far buffer (*data*) to the stream.

fwriteString

```
void fwriteString( const char far * str );
```

Writes the specified far character string (*str*) to the stream.

registerObject

```
void registerObject(PCvoid adr);
```

protected

Registers the class of the object pointed to by *adr*.

seekp

```
Ropstream seekp(streampos pos);
Ropstream seekp(streamoff off, seek_dir dir);
```

The first form moves the stream's current position to the absolute position given by *pos*. The second form moves to a position relative to the current position by an offset *off* (+ or –) starting at *dir*. *dir* can be set to *beg* (start of stream), *cur* (current stream position), or *end* (end of stream).

tellp

```
streampos tellp();
```

Returns the (absolute) current stream position.

writeByte

```
void writeByte(uchar ch);
```

Writes the byte *ch* to the stream.

writeBytes

```
void writeBytes(PCvoid data, size_t sz);
```

Writes *sz* bytes from *data* buffer to the stream.

writeData

```
void writeData(RTStreamable);
```

protected

Writes data to the stream by calling the appropriate class's **write** member function for the object being written.

See also: **TStreamable** and the **write** functions in the streamable classes

writePrefix

```
void writePrefix(RCTStreamable);
```

protected

Writes the class name prefix to the stream. The **<<** operator uses this function to write a prefix and suffix around the data written with **writeData**. The prefix/suffix is used to ensure type-safe stream I/O.

See also: **ipstream:readPrefix**

writeString

```
void writeString(PCchar str);
```

Writes *str* to the stream (together with a leading length byte).

writeSuffix `void writeSuffix(RCTStreamable);` **protected**

Writes the class name suffix to the stream. The **<<** operator uses this function to write a prefix and suffix around the data written with **writeData**. The prefix/suffix is used to ensure type-safe stream I/O.

See also: **ipstream:readPrefix**

writeWord `void writeWord(ushort us);`

Writes the word *us* to the stream.

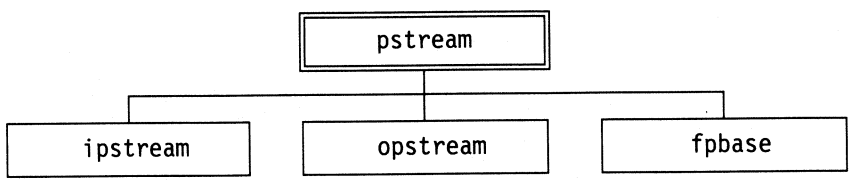
Friends

operator <<

```
friend Ropstream operator << (Ropstream  ps, signed char ch);
friend Ropstream operator << (Ropstream  ps, unsigned char ch);
friend Ropstream operator << (Ropstream  ps, signed short sh);
friend Ropstream operator << (Ropstream  ps, unsigned short sh);
friend Ropstream operator << (Ropstream  ps, signed int i);
friend Ropstream operator << (Ropstream  ps, unsigned int i);
friend Ropstream operator << (Ropstream  ps, signed long l);
friend Ropstream operator << (Ropstream  ps, unsigned long l);
friend Ropstream operator << (Ropstream  ps, float f);
friend Ropstream operator << (Ropstream  ps, double d);
friend Ropstream operator << (Ropstream  ps, long double d);
friend Ropstream operator << (Ropstream  ps, RTStreamable t);
friend Ropstream operator << (Ropstream  ps, PTStreamable t);
```

Inserts (writes) the given argument to the given **ipstream** object. The data type of the argument determines the form of write operation employed.

pstream objstrm.h



pstream is the base class for handling streamable objects.

Data members

bp `Pstreambuf bp;` **protected**

Pointer to the stream buffer.

state `int state;` **protected**

Format state flags. Use **rdstate** to access the current state.

See also: **pstream::rdstate**

types `static PTStreamableTypes types;` **protected**

Pointer to a database of all registered types in this application.

See also: **pstream::initTypes**

Member functions

constructor

```
pstream(Pstreambuf buf);
pstream();
```

protected

The first form creates a buffered **pstream** with the given buffer and sets the *bp* data member to *buf*. The state is set to 0. With the second form you can use **init** and **setstate**.

destructor `virtual ~pstream();`

Destroys the **pstream** object.

bad `int bad() const;`

Returns nonzero if an error occurs.

clear `void clear(int aState = 0);`

Set the stream *state* to the given value (defaults to 0).

eof `int eof() const;`

Returns nonzero on end of stream.

error

```
void error(StreamableError, RCTStreamable);
void error(StreamableError);
```

protected

Sets the given error condition, where **StreamableError** is defined as follows:

```
enum StreamableError { peNotRegistered, peInvalidType };
```

fail `int fail() const;`

Returns nonzero if a stream operation fails.

good `int good() const;`

Returns nonzero if no state bits set (that is, no errors occurred).

init `void init(Pstreambuf sbp);` **protected**

Initializes the stream: sets *state* to 0 and *bp* to *sbp*.

initTypes `static void initTypes();`

Creates the associated database object, pointed to by *types*. Called by the **TStreamableClass** constructor.

See also: **TStreamableClass**

operator void *() `operator Pvoid() const;`

Overloads the pointer-to-**void** cast operator. Returns 0 if operation has failed (that is, **pstream::fail** returned nonzero); otherwise, returns nonzero.

See also: **pstream::fail**

operator ! () `int operator ! () const;`

Overloads the NOT operator. Returns the value returned by **pstream::fail**.

See also: **pstream::fail**

rdbuf `Pstreambuf rdbuf() const;`

Returns the *pb* pointer to this stream's assigned buffer.

See also: **pstream::pb**

rdstate `int rdstate() const;`

Returns the current *state* value.

setstate `void setstate(int b);` **protected**

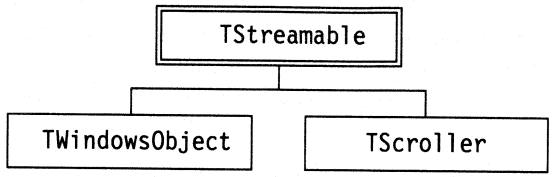
Updates the *state* data member with `state |= (b & 0xFF)`.

Friends

The private database class is a friend of **pstream**.

TStreamable

objstrm.h



Classes that inherit from **TStreamable** are known as streamable classes, meaning their objects can be written to and read from streams. If you want to develop your own streamable classes, you should make sure that **TStreamable** is somewhere in their ancestry. Using an existing streamable class as a base, of course, is an obvious way of achieving this. Don't be afraid to use multiple inheritance to derive a class from **TStreamable** if your class must also fit into an existing class hierarchy.

Since **TStreamable** is an abstract class, no objects of this class can be instantiated. Classes that inherit from **TStreamable** must redefine its three pure virtual functions **streamableName**, **read**, and **write**.

Member functions

read

```
virtual Pvoid read(Ripstream) = 0;
```

protected

This pure virtual member function must be redefined (or redeclared as pure virtual) by every derived class. The redefining **read** function for each streamable class must read the necessary data members from the supplied **ipstream** object. **read** is usually implemented by calling the base class's **read** (if any), then extracting any additional data members for the derived class.

See also: **TStreamableClass::ipstream**

streamableName

```
virtual const Pchar streamableName() const = 0;
```

private

This pure virtual member function must be redefined (or redeclared as pure virtual) by every derived class. Its purpose is to return the name of the streamable class of the object that invokes it. This name is used in the registering of classes by the stream manager. The name returned must be a unique, 0-terminated string (for example, "TDialog").

See also: **TStreamableClass, opstream, ipstream**

write `virtual void write(Ropstream) = 0;` **protected**

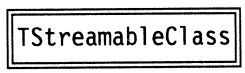
This pure virtual member function must be redefined (or redeclared as pure virtual) by every derived class. The redefining **write** function for each streamable class must write the necessary data members to the **opstream** object. **write** is usually implemented by calling the base class's **write** (if any), then inserting any additional data members for the derived class.

See also: **TStreamableClass, opstream**

Friends

The classes **opstream** and **ipstream** are friends of **TStreamable**.

TStreamableClass objstrm.h



TStreamableClass is used by the private database class and **pstream** in the registration of streamable classes.

Member function

constructor `TStreamableClass(PCchar n, BUILDER b, int d);`

Creates a **TStreamable** object with the given name (*n*) and the given builder function (*b*), then registers the type. Each streamable class ***TClassname*** has a**build** member function of type BUILDER (see Chapter 18, "Miscellaneous components"). For type-safe object-stream I/O, the stream manager needs to access the names and the type information for each class. To ensure that the appropriate functions are linked into any application using the stream manager, you must provide a reference such as:

```
TStreamableClass RegClassName;
```

where **TClassName** is the name of the class for which objects need to be streamed. (Note that *RegClassName* is a single identifier.) This not only registers **TClassName** (telling the stream manager which **build** function to

use), it also automatically registers any dependent classes. You can register a class more than once without any harm or overhead.

Invoke this function to provide raw memory of the correct size into which an object of the specified class can be read. Because the build procedure invokes a special constructor for the class, all virtual table pointers are initialized correctly.

macro, TStreamable _ _DELTA and d

The distance, in bytes, between the base of the streamable object and the beginning of the **TStreamable** component of the object is *d*. Calculate *d* by using the _ _DELTA macro (see Chapter 18, "Miscellaneous components"). For example,

```
TStreamableClass RegTClassName = TStreamableClass("TClassName",
TClassName::build, _ _DELTA(TClassName));
```

See also: **TStreamable, ipstream, opstream**

Friends

The classes **opstream**, and **ipstream** are friends of **TStreamableClass**.

C H A P T E R

18

Miscellaneous components

This chapter contains various ObjectWindows' parts that are not described in Chapter 16, "Class reference" or Chapter 17, "Streamable class reference."

The items listed in this chapter include types, constants, variables, macros, classes, and functions often declared in the ObjectWindows header files. A typical entry looks like the following **Sample**:

Sample — Sample's header file

Declaration `ReturnType Sample(ParamType AParam);`

Sample performs some useful function on its parameter, *AParam*.

See also **Example**

applicationClass constant — owldefs.h

Description A class identifier used to identify instances of the **TApplication** class. This constant is returned by the **isA** member function to identify the type of the object to the container classes.

See also **TApplication::isA**

BF_*XXXX* constants — owldefs.h

Description The following button flag constants are defined:

Table 18.1 Button flag constants

Constant	Value	Meaning
BF_UNCHECKED	0x0	Item is unchecked.
BF_CHECKED	0x1	Item is checked.
BF_GRAYED	0x2	Item is grayed.

Check box and radio button objects use BF_ constants to define their possible states.

See also **TCheckBox::SetCheck, TCheckBox::GetCheck**

BUILDER type — objstrm.h

Definition

```
typedef PTStreamable (_FAR *BUILDER)();
```

Used as the type for the build function parameter of the **TStreamableClass** constructor. Specifies the function called to build a template object when reading objects from a stream.

See also **TStreamableClass**

_CLASSDEF(classname) macro — _defs.h

Description _CLASSDEF is used at the top of header files that define classes. The macro defines the following typedefs for the class *classname*:

```
typedef classname _FAR * Pclassname;
typedef classname _FAR & Rclassname;
typedef classname _FAR * _FAR & RPclassname;
typedef const classname _FAR * PCclassname;
typedef const classname _FAR & RCclassname;
```

These typedefs are useful for DLL programming. The _FAR macro is automatically defined to be "far" when appropriate, so you don't need to explicity use it for data pointers and references.

_CLASSDLL macro

Description This macro is set by the user on the compiler command line (or in the IDE) to signify that the module being compiled will use classes in a DLL or will

pass classes into a DLL. Other macros (notably _CLASSTYPE, _EXPORT, and _FAR) check to see if _CLASSDLL has been defined.

_CLASSTYPE macro — _defs.h

Description Use this macro in the header file of a class that may be used with DLLs. It expands into either "huge", "far", or "near" as appropriate.

If the _ _DLL_ _ or _CLASSDLL macro is defined, _CLASSTYPE expands into "huge", since huge classes are required when they are passed between an EXE and a DLL, or between two DLLs. If the _ _DLL_ _ or _CLASSDLL macro is not defined, _CLASSTYPE expands as follows:

Table 18.2 _CLASSTYPE macro non-DLL expansion

Data model	Expansion	Pointer
tiny, small, medium	near	near **this**, near vtable
large, compact	far	far **this**, near vtable
huge	huge	far **this**, far vtable

CM_*XXXX* constants — owldefs.h

Description The following constants define start value or number of messages for user and internal command messages (CM):

Table 18.3 Command message constants

Constant	Value	Meaning
CM_COUNT	0x6000	Number of user CM
CM_FIRST	0xA000	Beginning of user CM
CM_INTERNAL	0xFF00	Beginning of reserved internal CM
CM_RESERVED	CM_INTERNAL – CM_FIRST	N/A

Following are the constant identifiers and classes with member functions that define these command-based messages:

Table 18.4: Command-based constants

Constant	Name	Common menu equivalent
CM_EDITCUT	**TEdit::CMEditCut**	Edit	Cut
CM_EDITCOPY	**TEdit::CMEditCopy**	Edit	Copy
CM_EDITPASTE	**TEdit::CMEditPaste**	Edit	Paste
CM_EDITDELETE	**TEdit::CMEditDelete**	Edit	Delete
CM_EDITCLEAR	**TEdit::CMEditClear**	Edit	Clear
CM_EDITUNDO	**TEdit::CMEditUndo**	Edit	Undo
CM_EDITFIND	**TEditWindow::CMEditFind**	Edit	Find
CM_EDITREPLACE	**TEditWindow::CMEditReplace**	Edit	Replace

Table 18.4: Command-based constants (continued)

CM_EDITFINDNEXT	**TEditWindow::CMEditFindNext**	Edit | Find | Next
CM_FILENEW	**TFileWindow::CMFileNew**	File | New
CM_FILEOPEN	**TFileWindow::CMFileOpen**	File | Open
CM_FILESAVE	**TFileWindow::CMFileSave**	File | Save
CM_FILESAVEAS	**TFileWindow::CmFileSaveAs**	File | Save As
CM_ARRANGEICONS	**TMDIFrame::CMArrangeIcons**	Window | Arrange Icons
CM_TILECHILDREN	**TMDIFrame::CMTileChildren**	Window | Tile
CM_CASCADECHILDREN	**TMDIFrame::CMCascadeChildren**	Window | Cascade
CM_CLOSECHILDREN	**TMDIFrame::CMCloseChildren**	Window | Close All
CM_CREATECHILD	**TMDIFrame::CMCreateChild**	Window | Create Child
CM_EXIT	**TWindowsObject::CMExit**	File | Exit

__DELTA macro

objstrm.h

Description Macro used to compute the offset, in bytes, from the beginning of a streamable object to the **TStreamable** component of that object. Its argument is the name of the streamable class.

See also **TStreamableClass**

dialogClass constant

owldefs.h

Description A class identifier used to identify instances of the **TDialog** class. This constant is returned by the **isA** member function to identify the type of the object to the container classes.

See also **TDialog::isA**

__DLL__ macro

Description Macro set automatically when compiling a DLL module, using –WD or –WDE switch. Other macros and conditional compilation use __DLL__ to determine whether a DLL is being built.

EM_XXXX constants — owldefs.h

Description The following error codes are defined:

Table 18.5 Error condition constants

Constant	Value	Meaning
EM_INVALIDCHILD	–1	One or more of the window's children is not valid.
EM_INVALIDCLIENT	–2	MDI client window could not be created.
EM_INVALIDMAINWINDOW	–3	Main window could not be created.
EM_INVALIDMODULE	–4	Window is invalid because Module object was not set.
EM_INVALIDWINDOW	–5	Window is invalid because **Create** did not succeed.
EM_OUTOFMEMORY	–6	A memory allocation ate into the safety pool.

EM_ constants are used to flag error conditions detected by ObjectWindows.

_EXPORT macro — owldefs.h

Description This macro conditionally expands into _EXPORT if a DLL is being built (__DLL__ is set). Otherwise it expands to _CLASSTYPE.

_FAR macro — _defs.h

Description Macro used to control the size of pointers in DLL data and code. DLL programs require far pointers that would otherwise be near (in small and medium models).

_FAR is defined to be "far" if either the __DLL__ or _CLASSDLL macro is defined. _FAR is used in parameter lists, return values, the far/near calling convention of functions and member functions, and class data members that are pointers. The typedefs in the following table provide a shorthand notation for some of the types frequently used where compatibility with DLLs is required.

Table 18.6: Typedefs, DLL-compatible

Type	Definition	Header file	Description
PCchar	`typedef const char _FAR * PCchar;`	clstypes.h	Pointer to a constant char
Pchar	`typedef char _FAR * Pchar;`	clstypes.h	Pointer to a char
PCvoid	`typedef const void _FAR * PCvoid;`	clstypes.h	Pointer to a constant of unspecified type
Pvoid	`typedef void _FAR * Pvoid;`	clstypes.h	Pointer to an unspecified type
Rint	`typedef int _FAR & Rint;`	owldefs.h	Reference to an int
RPvoid	`typedef void _FAR * _FAR & RPvoid;`	clstypes.h	Reference to a pointer of unspecified type
RTMessage	`typedef TMessage _FAR & RTMessage;`	windobj.h	Reference to a TMessage structure

GetApplicationObject function — applicat.h

Declaration

```
extern PTApplication _EXPFUNC GetApplicationObject();
```

Description

This function can be called from within an application or a DLL. If an ObjectWindows application is in control, **GetApplicationObject** returns a pointer to that application's application object (**PTApplication**). Otherwise, returns NULL.

ID_XXXX constants — owldefs.h

Description

The following constants define values or ranges for child-ID-based messages:

Table 18.7 Child ID message constants

Constant	Value	Meaning
ID_COUNT	0x1000	Number of child ID messages
ID_FIRST	0x8000	Start of child ID messages
ID_INTERNAL	0x8F00	Reserved for internal use
ID_RESERVED	ID_INTERNAL – ID_FIRST; N/A	

Default child identifiers follow:

Table 18.8 ID_command offset based values

Constant	Value	Meaning
ID_FIRSTMDICHILD	ID_RESERVED + 1	MDI child-ID
ID_MDICLIENT	ID_RESERVED + 2	MDI client window ID

ObjectWindows defines several categories of ID_ constants. One category contains constants that define the allowable ranges for child identifiers. A second category contains child identifiers that ObjectWindows uses as defaults.

_ _link macro — objstrm.h

Description Macro used to force the linker to link in the streamable object support needed to read or write a given class. The argument is the name of the **TStreamableClass** static registration object (the convention is *RegClassName* for a class named **TClassName**).

moduleClass constant — owldefs.h

Definition A class identifier used to identify instances of the **TModule** class. This constant is returned by the **isA** member function to identify the type of the object to the container classes.

See also **TModule::isA**

NF_*XXXX* constants — owldefs.h

Description The following constants are defined:

Table 18.9 Notification message constants

Constant	Value	Meaning
NF_COUNT	0x1000	Number of notification messages
NF_FIRST	0x9000	Beginning of notification messages
NF_INTERNAL	0x9F00	Beginning of notification messages reserved for internal use

NF_ constants define a range of locations for notify-based messages from 0x9000-0x9FFF, where 0x9F00-0x9FFF are reserved for internal use.

operator delete

Declaration `void operator delete (Pvoid ptr);`

ObjectWindows redefines the global **operator delete** to check the safety pool after each deletion. If it is exhausted, **delete** attempts to reallocate it. This provides automatic safety pool restoration whenever it is likely to be successful.

See also **SafetyPool::IsExhausted, operator new**

operator new

Declaration `Pvoid operator new(size_t s);`

ObjectWindows redefines the global **operator new** to use the safety pool. If an allocation fails, **new** deletes the existing safety pool if one is present. Then it attempts to allocate the object a second time. If this fails, 0 is returned.

Once the safety pool object has been freed, a low memory condition occurs. Future calls to **SafetyPool::IsExhausted** return 1.

See also **SafetyPool::IsExhausted, operator delete**

OWLGetVersion function owl.h

Declaration `WORD FAR OWLGetVersion();`

OWLGetVersion returns the version number of ObjectWindows, the value of the constant *OWLVersion*.

OWLVersion constant owl.h

Definition `const inst OWLVersion;`

You can use this constant to ensure that the version number returned from a call to **OWLGetVersion** is the expected version. This is most useful when using the ObjectWindows DLL.

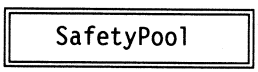
P_id_type typedef — objstrm.h

Definition

```
typedef unsigned P_id_type;
```

Used internally by the streamable object implementation to search the database of objects that have already been read or written.

SafetyPool class — safepool.h



This class provides an encapsulation of the safety pool mechanism used by ObjectWindows to ensure that interface objects are constructed in an atomic operation.

Data members

safetyPool

```
static Pvoid safetyPool;
```

This pointer to the safety pool object is set at run time to a block of memory of size equal to *Size*. Applications taking advantage of the safety pool to determine low memory conditions can determine whether the safety pool has been exhausted by calling **IsExhausted**. To change the size of the safety pool, use **Allocate**.

See also **SafetyPool::IsExhausted,** *SafetyPool::Size,* **SafetyPool::Allocate**

Size

```
static int Size;
```

Contains the current size of the safety pool, initialized by the class library at run time to the value of the constant DEFAULT_SAFETY_POOL_SIZE.

See also **SafetyPool::Allocate,** *SafetyPool::safetyPool*

Member functions

Allocate

```
static int Allocate(size_t size = DEFAULT_SAFETY_POOL_SIZE);
```

Deletes the existing safety pool object and reallocates one of the supplied size. If no size is specified, it defaults to the value of the constant DEFAULT_SAFETY_POOL_SIZE. Returns 1 if successful; otherwise, 0.

IsExhausted

```
static int IsExhausted();
```

This static member function returns 1 if the safety pool has been exhausted; otherwise, 0. The safety pool is considered to have been exhausted when an allocation has resulted in deallocating the safety pool, setting *safetyPool* to 0. When this condition has occurred, the low memory situation continues until such time as a successful call to **Allocate** is made.

See also **SafetyPool::Allocate**

scrollerClass constant — owldefs.h

Description A class identifier used to identify instances of the **TScroller** class. This constant is returned by the **isA** member function to identify the type of the object to the container classes.

See also **TScroller::isA**

SD_*XXXX* constants — owlrc.h

Description The following constants are defined.

Table 18.10 Standard dialog box constants

Constant	Decimal Value
SD_FILEOPEN	32512
SD_FILESAVE	32513
SD_INPUTDIALOG	32514
SD_SEARCH	32528
SD_REPLACE	32529

SD_ constants are the identifiers of ObjectWindows dialog boxes. Dialog resource templates for SD_SEARCH and SD_REPLACE are in STDWNDS.DLG.

StreamableInit type — objstrm.h

Definition

```
enum StreamableInit {streamableInit} ;
```

Streamable object support provides a unique constructor for each class for initializing objects which are read in from object streams. Each constructor has a parameter list of one argument of the enumeration type **StreamableInit**.

TActionFunc type — windobj.h

Definition

```
typedef void (_FAR * TActionFunc)(Pvoid Child, Pvoid Paramlist);
```

A **TActionFunc** function pointer is passed to **TWindowsObject::ForEach**.

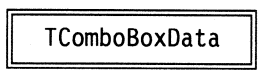
TActionMemFunc type — windobj.h

Definition

```
typedef void (TWindowsObject::* _FAR TActionMemFunc)(Pvoid Child, Pvoid
ParamList);
```

A **TActionMemFunc** member function pointer is passed to **TWindowsObject::ForEach**.

TComboBoxData class — combobox.h



A **TComboBoxData** is an interface object that represents a transfer buffer for a **TComboBox**.

Data members

Selection

```
Pchar Selection;
```

A pointer to the currently selected string, for transfer to or from a combo box.

Strings

```
PArray Strings;
```

Array of strings for transfer into the combo box.

Member functions

constructor

```
TComboBoxData();
```

Constructs a **TComboBox** object. Initializes *Strings* to an empty **Array** and *Selection* to NULL.

destructor

```
~TComboBoxData();
```

Deletes the space allocated for *Strings* and *Selection*.

AddString

```
void AddString(PChar AString, BOOL IsSelected = FALSE);
```

Adds the specified string to the array of *Strings*. If *IsSelected* is TRUE, deletes *Selection* before reallocating a copy of *AString* as *Selection*.

TCondFunc type — windobj.h

Definition

```
typedef BOOL (_FAR * TCondFunc)(Pvoid Child, Pvoid Paramlist);
```

A **TCondFunc** function pointer is passed to **TWindowsObject::FirstThat**.

See also **TWindowsObject::FirstThat**

TCondMemFunc type — windobj.h

Definition

```
typedef BOOL (TWindowsOject::* _FAR TCondMemFunc)
   (Pvoid Child, Pvoid Paramlist);
```

A **TCondMemFunc** member function pointer is passed to **TWindowsObject::FirstThat**.

See also **TWindowsObject::FirstThat**

TDialogAttr type — dialog.h

Definition

```
struct _CLASSTYPE TDialogAttr {
   LPSTR Name;
   DWORD Param;
};
```

A **TDialogAttr** is used to hold a **TDialog**'s creation attributes. The *Name* data member holds the identifier of the dialog resource. The *Param* data member holds a parameter that is supplied to the dialog box when it is

created. Windows accepts *Param* and it is then available in the message response functions associated with WM_INITDIALOG.

See also **TDialog::Attr**

TF_*XXXX* constants — owldefs.h

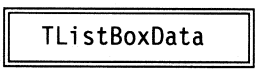
Description The following constants are defined:

Table 18.11 Transfer function constants

Constants	Value	Meaning
TF_GETDATA	0	Retrieve data from the class.
TF_SETDATA	1	Send data to the class.
TF_SIZEDATA	2	Return the size of data transferred by the class.

TF_ constants are passed to the **Transfer** member functions of window classes to indicate the operation to be performed.

TListBoxData class — listbox.h



A **TListBoxData** is an interface object that represents a transfer buffer for a **TListBox**.

Data members

SelCount

```
int SelCount;
```

The number of selected items.

SelStrings

```
PArray SelStrings;
```

Pointer to an array of the strings to select when data is transferred into the list box. When data is transferred out of the list box, *SelStrings* returns the current selection(s).

Strings

```
PArray Strings;
```

Pointer to an array of strings to be transferred into a **TListBox**.

Member functions

constructor

```
TListBoxData();
```

Constructs *Strings* and *SelStrings*. Initializes *SelCount* to 0.

destructor

```
~TListBoxData();
```

Deletes the space allocated for *Strings* and *SelStrings*.

AddString

```
void AddString(PChar AString, BOOL IsSelected = FALSE);
```

Adds the specified string to *Strings*. If *IsSelected* is TRUE, adds the string to *SelStrings* and increments *SelCount*.

GetSelString

```
void Get SelString(LPSTR Buffer, int BufferSize, int Index = 0);
```

Locates the string at the specified *Index* in *SelStrings* and copies it into *Buffer*. *BufferSize* includes the terminating NULL.

GetSelStringLength

```
int GetSelStringLength(int Index = 0);
```

Returns the length (excluding the terminating NULL) of the string at the specified *Index* in *SelStrings*.

ResetSelections

```
void ResetSelections();
```

Removes all strings from *SelStrings* and sets *SelCount* to 0.

SelectString

```
void SelectString(LPSTR AString);
```

Adds *AString* to *SelStrings*, incrementing *SelCount*.

TMessage type — windobj.h

Definition

```
struct TMessage {
   HWND Receiver;
   WORD Message;
   union {
      WORD WParam;
      struct tagWP {
        BYTE Lo;
        BYTE Hi;
      } WP;
   };
   union {
      DWORD LParam;
```

```
        struct tagLP {
          WORD Lo;
          WORD Hi;
        } LP;
      };
      long Result;
};
```

A **TMessage** structure contains information about an incoming message from Windows. The structures are filled by ObjectWindows and passed to message response member functions.

Receiver contains the handle of the window for which the message is originally intended. *Message* contains the message identifier. *WParam* and *LParam* members contain message-specific information retrieved from Windows. **LP** and **WP** structures are defined for easy access to the high and low order parts of *WParam* and *LParam*. *Result* contains the value to be returned to Windows.

TScrollBarData type — scrollba.h

Definition

```
struct TScrollBarData {
   int LowValue;
   int HighValue;
   int Position;
};
```

A **TScrollBarData** structure is used to store **TScrollBar** data. *LowValue* and *HighValue* are used to set and retrieve the range of the scroll bar. *Position* is used to set and retrieve the position of the scroll bar's thumb.

See also **TScrollBar::Transfer**

TSearchStruct type — editwnd.h

Definition

```
struct _CLASSTYPE TSearchStruct {
   char SearchText[81];
   BOOL CaseSensitive;
   char ReplaceText[81];
   BOOL ReplaceAll;
   BOOL PromptOnReplace;
};
```

A *TSearchStruct* holds search/replace text and related options passed by a **TSearchDialog**. The parameter names indicate their functions. *SearchText*

and *ReplaceText* contain user-input character strings. Other parameters are user-selected options for case sensitivity and replacement of all occurrences. *PromptOnReplace* indicates whether or not to prompt the user to confirm a replace operation.

See also **TSearchDialog::TSearchDialog, TEditWindow::CMEditFind, TEditWindow::CMEditReplace**

TWindowAttr type — window.h

Definition

```
struct _CLASSTYPE TWindowAttr {
   DWORD Style;
   DWORD ExStyle;
   int X, Y, W, H;
   LPSTR Menu;
   int Id;
   LPSTR Param;
};
```

A **TWindowAttr** is used to hold a **TWindow**'s creation attributes.

Style and *ExStyle* contain the window and extended styles. *X* and *Y* contain the screen coordinates of the top left corner of the window. *W* and *H* contain the width and height values of the window. *Param* contains a value that is passed to Windows when the window is created. It is then available in the message response functions associated with WM_CREATE. *Param* is used by **TMDIClient** and may be useful when converting non-ObjectWindows code.

Menu contains the resource identifier of the window's menu (if any). *Id* contains the identifier of the window; for a dialog box control *Id* is its resource identifier.

See also **TWindow::Attr**

WB_*XXXX* constants — owldefs.h

Description The following values are defined:

Table 18.12 TWindowsObject attribute masks

Constant	Value	Meaning of corresponding attribute, if set
WB_ALIAS	0x01	Associated with window in caller of a DLL.
WB_AUTOCREATE	0x02	Created along with parent.
WB_FROMRESOURCE	0x04	Created from a resource definition.
WB_KBHANDLER	0x08	Provides keyboard interface to controls.
WB_MDICHILD	0x10	Is an MDI child window.
WB_MDIFRAME	0x20	Is an MDI main (frame) window.
WB_PREDEFINEDCLASS	0x40	Has a predefined Window's class.
WB_TRANSFER	0x80	Participates in **Transfer**.

WB_ constants define bit masks for the internally used flag attributes of **TWindowsObject**. A WB_ mask is defined for each of these attributes.

See also **TWindowsObject::SetFlags, TWindowsObject::EnableAutoCreate, TWindowsObject::EnableKBHandler, TWindowsObject::EnableTransfer**

windowClass constant — owldefs.h

Description A class identifier used to identify instances of the **TWindow** class. This constant is returned by the **isA** member function to identify the type of the object to the container classes.

See also **TWindow::isA**

WM_*XXXX* constants — owldefs.h

Description The following constants are defined:

Table 18.13 Window message constants

Constant	Value	Meaning
WM_COUNT	0x8000	Number of reserved messages
WM_FIRST	0x0000	Beginning of reserved range
WM_INTERNAL	0x7F00	Beginning of ObjectWindows reserved range
WM_RESERVED = WM_INTERNAL-WM_FIRST		N/A

ObjectWindows defines constants related to the Windows WM_XXXX messages. The range 0x7F00-0x7FFF is reserved for internal ObjectWindows use.

I N D E X

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z